CACTVS Tcl Scripting Introduction

Script Interpreter Programs

The standard distribution of the toolkit contains a number of standard application scripts, and two general-purpose interpreters for the execution of custom scripts.

Start scripts

When the setup program is executed, a number of start scripts named cs?? are generated. The prefix *cs* is the abbreviation for **CACTVS** System, and the final two or three letters are a short mnemonic form of the application, such as *br* for *structure browser*, or *tb* for *table builder*. Most of these start scripts execute a predefined script, but two of them are general-purpose interpreters which can be fed with any script.

On Unix, these start scripts are short shell scripts which set up a number of environment variables and then start one of two general-purpose chemistry interpreters. One of these is *tclcactvs*, the other *tkcactvs*. The only difference in functionality is that the latter includes the Tk platform-independent GUI toolkit. In principle, it is also possible to start with a plain *tclcactvs*, and then load Tk as a toolkit at run-time.

On MS Windows, the cs?? files are short-cuts which start *tclcactvs* or *tkcactvs* with the application script file directly. On Windows, path information is extracted from the registry, so no environment set-up is required.

This is a sample start script for the generic GUI-less script interpreter *csts*:

```
#! /bin/sh
TK_LIBRARY=/usr/local/lib/cactvs/tk8.2
TCL_LIBRARY=/usr/local/lib/cactvs/tcl8.2
TKX_LIBRARY=/usr/local/lib/cactvs/tkx8.2
TCLX_LIBRARY=/usr/local/lib/cactvs/tclx8.2
BLT_LIBRARY=/usr/local/lib/cactvs/blt2.4
TIX_LIBRARY=/usr/local/lib/cactvs/tix4.1
OS="Linux2.4"
CACTVS_DATA_DIRECTORY=/usr/local/lib/cactvs
PATH=$CACTVS_DATA_DIRECTORY:$PATH
LD_LIBRARY_PATH=/usr/local/lib/cactvs/lib:$LD_LIBRARY_PATH
LD_LIBRARYN32_PATH=/usr/local/lib/cactvs/lib:$LD_LIBRARYN32_PATH
export LD_LIBRARY_PATH LD_LIBRARYN32_PATH TKX_LIBRARY TCLX_LIBRARY
export TK_LIBRARY TCL_LIBRARY BLT_LIBRARY TIX_LIBRARY OS CACTVS_DATA_DIRECTORY
PGM=/usr/local/lib/cactvs/lib/tclcactvs
exec $PGM -b -d "$@"
```

The generic multi-platform version of this script which is part of the development distributions looks slightly more difficult, but does essentially the same. That script version determines the operating system version at runtime by analysing *uname* command output, and compensates for incompatibilities between SuSE and RedHat C++ runtime library naming conventions. The *uname* output analysis in the generic version looks complicated because it distinguishes between Linux on I386 and Alpha processor architectures, which are not trivial to keep apart from its messages.

If the environment variables are already set up appropriately, the start scripts are not needed and the central program *tclcactvs* and *tkcactvs* may be started directly.

Environment variables

The following environment variables are used by the **CACTVS** system:

- **`BLT_LIBRARY`** Path to the BLT runtime script library. Usually, it is a subdirectory of the **`$CACTVS_DATA_DIRECTORY`** directory. BLT is an extension module for the TK toolkit. All standard GUI applications included with the toolkit use this package, for example for the drag&drop functionality. This variable is only used by Tk-enabled interpreter versions. BLT is not compiled into the *tkcactvs* application, but loaded as a package on demand.
- **`CACTVS_DATA_DIRECTORY`** The variable stores the path to the directory which contains the **CACTVS** script libraries and other directories with critical runtime information. In standard customer installations, this is the base library installation directory, for example */usr/local/lib/cactvs*.
- **`LD_LIBRARY_PATH`** This variable is not used by the toolkit directly, but nevertheless very important because the toolkit is very modular, and many programs such as tkcactvs depend on a dozen or more dynamic libraries. The standard start-up scripts will prefix the library path with the location of the runtime libraries. On IRIX systems, the variable **`LD_LIBRARYN32_PATH`** is also important. The toolkit libraries and executables are compiled in *n32* format on all IRIX releases which support this linker format.
- **`OS`** The operating system. This variable will actually be re-exported by the initialization routine if it is not set. This variable is used as a path component in various **CACTVS** extension modules search paths in multi-platform installations. Setting this variable is optional, since it will be automatically generated if necessary. The syntax of this variable is the basic OS name as reported by the *uname* command, followed by its version, followed by the processor class if it is not the standard class. Examples of valid values are *SunOS5.10-64* (*not* Solaris something), and Linux2.6. On Windows, the generic OS name is *WIN32*.
- **`PATH`** This is the standard search path for executables. The start scripts will prefix the path with the directory where the executable programs of the toolkit are stored. Indirectly, this variable is used whenever an external executable is started by a script, or directly by the core toolkit code. The latter happens for example when I/O to or from pipes is performed, or when reading compressed structure files. An up-to-date *gzip* program should be installed on the system in a location contained in the **`PATH`** in order to enable input of structure files in standard compression formats.
- **`TCL_LIBRARY`** Path to the Tcl runtime script library. Usually, it is a subdirectory of the **`$CACTVS_DATA_DIRECTORY`** directory.
- **`TCLX_LIBRARY`** Path to the TclX runtime script library. Usually, it is a subdirectory of the **`$CACTVS_DATA_DIRECTORY`** directory. The TclX Tcl extension module is a compiled-in component of all Tcl-enabled toolkit versions. It is essential because it for example provides the functions for the encoding and decoding of object handles.

- **`TIX_LIBRARY`** Path to the Tix runtime script library. Usually, it is a subdirectory of the **`$CACTVS_DATA_DIRECTORY`** directory. Tix is another extension for the **Tk** GUI toolkit. It is used by some of the GUI application scripts, such as the structure browser *csbr*. However, not all GUI applications rely on this extension module. This variable is only used by **`Tk`**-enabled interpreter versions, and only of the Tix package has been loaded. Tix is not compiled into the *tkcactvs* application, but loaded as a package on demand.
- **`TK_LIBRARY`** Path to the **Tk** runtime script library. Usually, it is a subdirectory of the **`$`** directory. This variable is only used by **`Tk`**-enabled interpreter versions.

Stand-alone applications

The compilation environment of the toolkit supports the compilation of stand-alone executables. In these executables, the application script, all runtime libraries, script libraries, extra property definitions and toolkit extension modules are collected and linked into a big static executable. In addition, the dynamic extension capabilities of the toolkit are disabled in these programs, so that they will never attempt to load additional functionality from modules or description files.

These stand-alone applications do not access any environment variables, and the only installation procedure required is to move them into a standard binary directory, such as */usr/local/bin.*

From a user standpoint, stand-alone executables and application scripts started via a start script and backed by a full toolkit installation are supposed to be indistinguishable. The only notable difference is the behaviour in case unknown data is encountered, where the toolkit version will go through its procedure of automatic extension look-up and loading, while the stand-alone versions will not. Given a suitable set of additional property definition and extension modules compiled into the stand-alone versions, these programs have no need not have to refer to automatic loading of extra modules and their behaviour is indeed identical to the script version.

Encrypted scripts

It is possible to distributed confidential scripts and open scripts in the same environment. The script interpreters contain a compiled-in and usually customer-specific key which can be used to run encrypted scripts.

Encrypted scripts can be generated from a standard readable Tcl or Tk script by performing an RC4 encoding:

```
rc4 the_key <plain_script.tcl >enc_script.rc4
```

The encrypted script may be run on interpreters which have the same key by setting the -x option:

```
tclcactvs -x -f enc_script.rc4
```

Keys may also be set by providing a digitally signed *license.dat* file as part of a distribution. This file overrides the standard license settings, including the decryption key.

CACTVS as generic Tcl extension module

The standard **CACTVS** script interpreters contain the Tcl scripting language interface as a compiled-in component.

However, it is also possible to compile the **CACTVS** toolkit libraries without the main Tcl/Tk libraries, but still with Tcl scripting support. The basic toolkit library may then be loaded into any program which uses a Tcl interpreter as a Tcl module, similar to the way **CACTVS** uses the Tix and BLT extension modules. After loading, the Tcl extensions provided by **CACTVS** are available in the host application as additional commands. Since **CACTVS** does not know anything about the internal data structures of the host application, there will be no built-in mechanisms for direct interfacing with the host application data structures. However, data exchange on the Tcl level (such as passing a SMILES string which was generated by **CACTVS** commands) into a host application command is possible and useful.

Examples:

```
package require Cactvs
```

```
load /the/path/libcactvs.so
```

This are two methods to load the Cactvs library into a host application. The first example requires a suitable set-up of the host application *pkgIndex.tcl* file which contains the path information to the shared libraries of available extension modules. The second example is a direct load. In both cases, the environment variables which can influence the operation of the toolkit must have been set-up properly, especially the `CACTVS_DATA_DIRECTORY` variable.

In both cases, Tclx must be available as a module for the host application, or compiled into it. The Tcl scripting capabilities of **CACTVS** depend on this extension. When the `Cactvs` subsystem is initialized, an attempt will be made to load the Tclx package. If it was compiled in, or loaded into the host application before the **CACTVS** module as accessed, this will automatically succeed. Otherwise, the Tcl library will try to locate this package via its current package path. The best method to ensure that it is found is therefore to enter the load data for the Tclx package into the host application *pkgIndex.tcl* file if the host application does not already contain it as a compiled-in component.

This approach works with a number of different applications, starting with a plain Tclx interpreter (usually named *tcl* as executable file), via the Tcl Web browser plug-in to large packages such as the VMD visualization and modelling suite.

Toolkit libraries without Tcl scripting language support

Versions of the `Cactvs` toolkit are available which do not contain Tcl scripting language support. Since the core library and the scripting language environment are implemented as clearly separate layers, the removal of the scripting language layer (and potentially its replacement by other language bindings) is relatively painless.

A toolkit library without scripting language support must be programmed by calling C functions. The most important C functions for interfacing are described in a separate library programming document.

Examples for this kind of libraries are DLLs or shared object libraries based on the Cactvs toolkit which provide reduced, specialized functionality, for example for structure depiction or substructure searching. Generally, these libraries include all functionality, such as property definitions and modules, as compiled-in components and do not require any environment set-up or loading of additional modules.

Interpreter program options

Both the standard script interpreter *tclcactvs* and the GUI-enabled variant *tkcactvs* understand a number of options which can be used to modify the script execution.

For *tclcactvs*, these are:

• -!	Enable structure security. If this flag is set, structure and reaction information will not be sent over the Internet for computations which invoke external Web services. For academic distributions, this flag is off by default. Commercial packages have it enabled by default.
• -<*inputdirs*	A colon-separated list of directories where the application is allowed to read files. If this option is used, toolkit commands which open files for input outside the restricted directory set will raise an error.
• ->*outputdirs*	A colon-separated list of directories where the application is allowed to write files. If this option is used, toolkit commands which open files for output outside the restricted directory set will raise an error.
• -a	If set, the application writes a *csconfig.xml* file with the currently configured program set-up on exit.
• -b	Remove all compiled-in paths from the search paths for extension modules which are dependent on the build environment.
• -c*cmd*	Execute the specified command and exit immediately afterwards.
• -C*secs*	Set a CPU time execution limit. By default, no such limit is enforced.
• -d	Allow the loading of dynamic extensions. By default, only dynamic extension loading from trusted locations, such as the installation directory, are allowed. With this option, additional privately configurable locations which are not in the trusted path will be checked if an extension cannot be found in a standard location.
• -D	Similar to option *-d*, but with this option the additional non-trusted private locations will be searched *before* those in the standard path.
• -e	Run scripts with command tracing. Equivalent to executing `cmdtrace on` as first script command.
• -f*scriptfile*	Execute the script file. When the script finishes, the program exits. By default, if neither the *-f*, *-c*, or *-u* options are set, an interactive command line interpreter is started.
• -F*scriptfile*	Secondary script file sent to the interpreter of the language not executing the primary script (i.e. this would be a Python script if the primary script is **Tcl**. and vice versa).

- -h — Print short option help text and exit.
- -i0/1/2 — Control Internet access. Level 0 disables all automatic Internet look-up. Level one allows this in computational modules and file I/O modules, but this is further subject to the settings of the host control variables and the structure security flag. Level two enables Internet look-up also for property definitions and modules from Internet sources listed in the search paths.The default level is one. If the level is zero, all Internet host control variables will also be reset to `NULL`, so that even increasing the look-up level value in a script will not directly re-enable Internet look-ups without also re-initializing the host variables.
- -I — If set, enable *readline* support on interactive shells. This option is not supported on Windows.
- -k — Run a **KNIME** node I/O background thread on the default port 16570.
- -K*port,maxtime,maxrowstotal,maxrowsperport,maxfilesize,user,password,rpclogfile*
  Run a **KNIME** node I/O background thread with the specified options. Empty individual parameters (i.e. no text between two commas) are skipped and do not set empty or zero values.
- -l *configfile/configdir*
  Add a custom directory location or file to the start-up configuration processing. The custom file is processed last and its contents supersede configuration parameters set in other configuration files. If a directory name is used, the configuration file is expected to be named *csconfig* or *.csconfig*. Configuration files are searched, in this order, in the installation base directory, the user home directory, the current directory at start-up time (possibly modified by a -w option) and finally, if specified, the custom location.
- -L*urn* — Define local **URN** namespace.
- -M*limit* — Specify the maximum amount of memory to be used. The argument is an integer, optionally with a 'K' or 'M' suffix to indicate kilobytes or megabytes, respectively. By default, no memory limit is enforced.
- -N — If the program is running as a compute server, do not `fork()` the program for the processing of the request. Instead, further requests are blocked until the current request has been finished.
- -o — Use pre-resolved object pointers in **Tcl** value objects. This option can speed up scripts with lots of minor object references by up to 20%, but it will have subtle effects on the re-use of object handle names etc. In general, it is a safe option and most scripts will work without any change under this execution model. However, implicit assumptions about object naming schemes valid for standard scripts may no longer be guaranteed, and there is a risk of breaking code which is not cleanly written.

• -O*objlimit*	Set an chemistry object limit. If the number of active chemistry major objects in the application exceeds this number at any time, the program is terminated, presumably before it starts excessive swapping. By default, no object limit is enforced.
• -p*port*	Set RPC server port. Only ports beyond 1023 (the reserved space) are allowed.
• -P	If set, the main interpreter is forced to be a Python interpreter. By default the main interpreter type is defined by the suffix of the primary script, or the name of the executable if no script is processed.
• -q	Quick start-up without the loading of any extension modules.
• -r	Allow the processing of **RPC** requests, in addition to script execution or the interactive command line.
• -R	Configure for the exclusive processing of **RPC** or **KNIME** node server requests. Do not attempt to run a script or to start with a command line.
• -s*propertylist*	Configure to act as a compute server for the properties in the list (comma-separated). Only base property names following the **CACTVS** nomenclature are allowed, no subfields or original names. This option alone will not yet make the interpreter running as a server. It will only work if the *-r* or *-R* options are added. Property computation requests for properties which are not in the list will be rejected, as well as requests which fail the remote access check encoded as part of the property definition.
• -S*id*	Set the **RPC** service ID. This is an expert debugging feature.
• -t*value*	Activate subsystem traces. The option value is an unsigned integer. Every set bit in this number activates the tracing of a different subsystem, which are linked to a different bit position. The subsystems are listed below. Trace output is written to the *stderr* channel on Unix, and uses the **ATLTRACE** functions on Windows.
• -T	Disabled all computation or query time-out and signal processing. This is primarily useful when analysing the interpreter with a debugger and source code access.
• -u*url*	Execute a script which is downloaded from a URL. Currently, the toolkit understands **HTTP**, **FTP**, **Gopher** and **FILE** URLs.
• -U*username*	Run the program and script as the named user and in its primary group. This will only work if the program is started as root, and only on Unix.
• -v	Print software version and licensing information and then exit.
• -V*id*	Set the RPC program version. This is an expert debugging feature.
• -w*dir*	Change the working directory of the program to the specified directory before any scripts are executed.

• -x	Assume script file is encrypted and use compiled-in key to decode it transparently in memory when it is executed.
• -z*nsecs*	Sleep a number of seconds before commencing script and **RPC**-type operations. This is primarily useful for debugging, for example for attaching a debugger to a running toolkit process.
• -Z*certfile[:keyfile]*	Specify **SSL** server certificate and key files in **PEM** format. These are only needed when operating as a **KNIME** node server with support for encrypted communication. If no separate key file is specified, it is assumed to reside in the same directory as the certificate file, with the same base file name, but with suffix *.key*. If this parameter is not used, and the installed package contains a *knime* subdirectory, the default *localhost* certificate and key found in that directory are automatically configured.

These options can be used both with the standard start script *csts* as with the raw *tclcactvs* application. To distinguish between standard interpreter options and application options, the option separator -- can be used. Everything after this separator is passed as application options.

Example:

```
csts -q -f myscript.tcl -- -q myscript.dat
```

If called like this, the first *-q* will be consumed and used as an interpreter option, as is the *-f* specification, while the second *-q* parameter is passed to the script environment. For the script, the program arguments, which are stored in the standard global Tcl list variable *argv*, seem to consist of only two list elements with values "*-q*" and "*myscript.dat*", which are read from the argument list after the separator.

The arguments of options which require additional information, such as a file name, may be specified with or without a white space separator between the option character and the argument. Multiple option characters without arguments may be merged, for example as

```
csts -qfmyscript.tcl -- -q myscript.dat
```

which is completely equivalent to the sample line above.

The *tkcactvs* program options a little bit more limited:

• -console	Start a console window for controlling the main interpreter in a separate window. On Windows, this happens automatically if the program is started without a script. On Unix, the program displays a command line prompt in the start shell if not run with a script via the *-f* or *-u* options.
• -crypt	Corresponds to option *-x* of *tclcactvs* interpreter.
• -dynload	Corresponds to option *-d* of *tclcactvs* interpreter.
• -file *file*	Corresponds to option *-f* of *tclcactvs* interpreter.
• -internet	Corresponds to option *-i* of *tclcactvs* interpreter.
• -quick	Corresponds to option *-q* of *tclcactvs* interpreter.
• -namespace *ns*	Corresponds to option -L of *tclcactvs* interpreter.

-nobuildpath	Corresponds to option *-b* of *tclcactvs* interpreter.
-norpc	The inverse of option *-r* of tclcactvs interpreter.
-trace *bits*	Corresponds to option *-t* of *tclcactvs* interpreter.
-slave	Install a fully chemistry-enabled slave interpreter of the main interpreter. This is required for the editing of property computation scripts in the property editor *cspe*.
-url *url*	Corresponds to option *-u* of *tclcactvs* interpreter.
-wrapper *name*	Pass the name of a wrapper program which was used to invoke the interpreter.

In this application, the standard Tk argument parsing routines are used. Therefore, it is possible to abbreviate the full option name to the shortest unique part. For example, using *-f* is equivalent to the full option name *-file*. This application also understands the -- option list separator.

Codes for traceable subsystems

These are the codes for traceable subsystems. Multiple subsystems can be traced in parallel by summing up the codes and using this number for the -t (*tclcactvs*) or -trace (*tkcactvs*) program options:

(1<<0)	RPC communication
(1<<1)	General property calculation
(1<<2)	System initialization
(1<<3)	Binary I/O of native *Cactvs* formats
(1<<4)	Filter processing
(1<<5)	Database operations
(1<<6)	File I/O
(1<<7)	Stereochemistry
(1<<8)	Aromaticity resolution
(1<<9)	Object deletion
(1<<10)	Connectivity generation from 3D coordinates
(1<<11)	Client/server based property computation
(1<<12)	Basic ring system analysis
(1<<13)	Radical resolver
(1<<14)	Path walking
(1<<15)	Charge resolver
(1<<16)	Substructure matching
(1<<17)	Table operations
(1<<18)	2D layout coordinate generation
(1<<19)	File query operations
(1<<20)	Tree walking

• (1<<21)	Native **Cactvs** structure/reaction database queries
• (1<<22)	Extended ring sets
• (1<<23)	3D substructure match operations
• (1<<24)	Hydrogen addition and removal
• (1<<25)	Structure hash code computation
• (1<<26)	Timing and time-outs
• (1<<27)	SMILES/SMARTS decoder
• (1<<28)	Temporary debugging
• (1<<29)	Dynamic extension module loading
• (1<<30)	Query parsing and execution
• (1<<31)	Reaction processing

Registry entries on Windows

When the toolkit is installed on Windows, the registry entries listed in the table are set in **HEKY_LOCAL_MACHINE**. {app} is a place holder for the application installation directory. These registry settings associate the file suffixes *.tcl* with *tclcactvs.exe* and *.tk* with *tkcactvs.exe*. Stand-alone applications and special-purpose DLL libraries do not create registry entries, nor do they depend on them.

Path	Name	Value
Software\Xemistry		
Software\Xemistry\CACTVS	basedir	{app}
Software\Xemistry\CACTVS	blt_library	{app}\blt3.0
Software\Xemistry\CACTVS	datadir	{app}
Software\Xemistry\CACTVS	tcl_library	{app}\tcl8.6
Software\Xemistry\CACTVS	tclx_library	{app}\tclx8.4
Software\Xemistry\CACTVS	tix_library	{app}\tix8.4
Software\Xemistry\CACTVS	tk_library	{app}\tk8.6
Software\CLASSES\.tcl		tclFile
Software\CLASSES\.tk		tkFile
Software\CLASSES\tclfile		
Software\CLASSES\tclfile\DefaultIcon		
Software\CLASSES\tclfile\Shell		
Software\CLASSES\tclfile\Shell\Open		
Software\CLASSES\tclfile\Shell\Open\Command		"{app}\lib\tclcactvs.exe" -f %1 -- %s
Software\CLASSES\tkfile		
Software\CLASSES\tkfile\DefaultIcon		

Software\CLASSES\tkfile\Shell		
Software\CLASSES\tkfile\Shell\Open		
Software\CLASSES\tkfile\Shell\Open\Command		"{app}\lib\tkcactvs.exe" -f %1 -- %s

Standard Tcl and Tk Packages

CACTVS uses in its script interpreters and application scripts a number of auxiliary Tcl and Tk packages, which are also part of the standard distributions. In addition, any other Tcl or Tk package from third parties may be loaded into a **`Cactvs`** interpreter by the native Tcl mechanism (**`load`** and **`package`** commands).

This is the list of currently used packages:

Package	Description	Use in *tclcactvs*	Use in *tkcactvs*
Gd 2.0	Tk extension - pixel image generation module	Loadable as package	Used by *csimg* sample application. This Gd package is significantly extended compared to the standard Gd package.
Gdbm 1.10	**GDBM** file support module	Loadable as package	Loadable as package
Itcl 3.4	Object-oriented Tcl extension	Compiled-in	Compiled-in
Itk 3.4	Object-oriented Tk extension	Loadable as package	Compiled-in
Ldap1.0	Tcl module - communication with LDAP servers	Loadable as package	Loadable as package
Tc	**TOKYOCABINET** file support module	Loadable as package	Loadable as package
Tcl 8.6.1	Core Tcl scripting language interpreter	Compiled-in	Compiled-in
TclBlt3.0	Tcl extension - various additional functions	Loadable as package	Automatically loaded as package by standard start-up script
TclReadline 2.1.0	Interactive *readline* command line support	Loadable package, automatically loaded when Tcl interpreter is started with -I option	Loadable package
TclX 8.4	Tcl extension, provides object handle functionality	Compiled-in	Compiled-in

Thread 2.7.0	Tcl thread package	Compiled-in in some interpreter versions, loadable module in others. Automatically loaded when commands such as **`dataset addthread`** are executed.	Compiled-in in some interpreter versions, loadable module in others
Tix 8.4	Tk extension - various additional widgets	Loadable as package	Loaded by some application scripts as package
Tk 8.6.1	Core Tk GUI toolkit package	Loadable as package	Compiled-in
TkBlt3.0	Tk extension - various additional widgets, drag & drop	Loadable as package	Automatically loaded as package by standard start-up script
Tktable 2.10	Tk extension - table widget	Loadable as package	Used by QSAR table application *csqt,* loaded as package

Major Object Commands

Most of the toolkit functionality is linked to a small collection of objects, providing means to manipulate data and structure of these objects. The core set of objects are called *major objects*. Major objects may carry property data, and they may control *minor objects*, which cannot exist without a controlling major object. A major object is addressed in the scripting language environment by its handle.

The **CACTVS** toolkit currently supports the following major chemistry objects in its standard edition:

- **ens** molecular ensembles
- **reaction** reactions, consisting of reagent and product ensembles, and optionally also solvents, catalysts, etc.
- **molfile** chemical structure files of any type, not just MDL Molfiles
- **table** data tables, optionally associated with ensembles or reactions
- **dataset** dataset usually consisting of of ensembles or reactions, but can also hold networks and tables
- **network** universal graph object with nodes and connections for the modelling of complex data relationships

Each object class is associated with a Tcl command which implements the scriptable functionality the object class supports. The name of the command is the same as the object class it represents. The general structure of these commands is

objecttype subcommand objecthandle ?args?

When they are created, major objects are automatically assigned an object handle, which is a short identifier string. The identifier starts with the name of the object class, followed by a number.

Example: *ens0* is an ensemble handle.

When an object is destroyed, its handle becomes invalid, and their use will result in an error. However, object handles will be reused in an unpredictable fashion if new objects are created.

Example:

```
set ehandle [ens create CCC]
ens delete $ehandle
```

will first create an ensemble object from a SMILES string, and then delete it, showing the generic command syntax for major object commands.

Minor Object Commands

Major objects may contain minor objects. Minor objects are also chemistry objects, but they do not have their own handle. They do not exist outside the context of a major object.

Minor objects are usually identified by a combination of their major object handles and a numeric label. For each minor object class, a default label property exists. Its name is the standard object class property prefix, followed by `_LABEL`.

Examples:

```
A_LABEL, M_LABEL, G_LABEL, V_LABEL
```

Standard minor objects are:

- **atom** - subobject of ensembles
- **bond** - subobject of ensembles
- **connection** - subobject of networks
- **group** - subobject of ensembles (arbitrary collection of atoms, also recursively including other groups)
- **mol** - subobject of ensembles
- **pi** - subobject of ensembles (π system)
- **ring** - subobject of ensembles
- **ringsystem** - subobject of ensembles
- **sigma** - subobject of ensembles (σ bond system)
- **surface** - subobject of ensembles (surface elements, optionally related to atom)
- **vertex** - subobject of networks

Each minor object has a command which is associated with it. The name of the command is the same as the object class.

Examples for these commands:

```
atom get $ehandle 1 A_ELEMENT
bond atoms $ehandle 2
```

Not all subobject classes may be present at all times for any major object. For example, if no ring information was ever used during the history of an ensemble, the ring subobject list will not have been initialized. Whenever a property which is associated with a given type of subobject is requested, the system will check if the subobject class has been set up for the controlling major object and attempt to set it up.

Example:

```
ens get $ehandle R_SIZE
```

may automatically invoke the determination of ring systems if they have not yet been set up. For some subobject classes, for which no computational procedure exists (such as **group**), the initialization will set up an empty list. Automatic non-void set-up routines exist for rings, ringsystems, molecules, bonds, π systems and σ systems. In the case of bonds, an attempt will be made to reconstruct a bonding scheme from atomic 3D coordinates, if these are available.

Subobjects may be individually created and discarded. For some cases, such as groups, atoms, or bands, this makes sense. For rings, molecules, etc. this is likely to interfere with the system operations.

Subobject manipulations will have effects on the status of other subobject sets on the same major object. For example, the deletion of a ring atom will remove all groups where the atom was a member, and totally destroy all ring and ringsystem information.Introduction to Properties

Data on chemical objects is stored as property data. Every data item is associated with a property description. Property descriptions may be built-in, loaded from standard installation locations, (depending on the set-up) loaded from Internet sources and databases, or generated ad-hoc by scripting language commands.

CACTVS does not distinguish between built-in property descriptions or those from any other source. It has no fixed data structure for chemical objects, such as a standard set of properties it maintains. For the system, it does not matter if at any given time element information on atoms is stored as a periodic system number, an element symbol, or both. Any property data may be requested at any time. The system will try to find a way to compute the requested data from what is already known about the system by looking at the property definitions and associated computational methods.

Every property is associated with a specific object class - major or minor. Data for this property can only be attached to objects matching that class. The object class a property is related to can be decoded from its initial character. These are the prefixes for the most common object classes:

- `A_` atoms (of ensembles)
- `B_` bonds (of ensembles)
- `C_` connections (of networks)
- `D_` datasets
- `E_` ensembles
- `F_` structure files

- `G_` groups (of ensembles)
- `M_` molecules (of ensembles)
- `N_` networks
- `O_` surface elements (of ensembles, mnemonic: German Oberfläche means surface)
- `P_` π systems (of ensembles)
- `R_` rings (of ensembles)
- `S_` σ systems (of ensembles)
- `T_` tables
- `V_` vertices (of networks)
- `X_` reactions
- `Y_` ringsystems (of ensembles)
- Z_ table cell data (internal use only)

Examples:

A_LABEL	Atom property
X_IDENT	Reaction property

Property data on minor objects is always maintained and updated as a block.

Example:

Either all atoms in an ensemble have valid property data for `A_SYMBOL`, or none has.

Property Validity

Property data attach to an object will remain valid once it has been set until either

- The major object is destroyed
- It is explicitly deleted, overwritten, or re-computed
- The major object undergoes a modification which renders the data invalid
  Example: Atom deletion will delete molecule weight data. This applies also to major objects: Many dataset properties will probably become invalidated when an ensemble is removed from a dataset.
- The values of another property at the same major object change and that property has been listed as input data for the property considered.
  Example: If the molecular weight `M_WEIGHT` changes, the ensemble weight `E_WEIGHT` is invalidated, since the `E_WEIGHT` property definition declares it is dependent on `M_WEIGHT` (all it does in its computation routine is to sum up `M_WEIGHT` for all molecules).

The behaviour of property data with respect to object or data modifications is part of their property definition record. They may be completely oblivious to any such change, or very sensitive.

Hydrogen addition and removal with the standard commands is treated somewhat different from normal atom and bond manipulation. Since it is assumed that the hydrogen removal is only done for export purposes, to prepare the data for software which still uses the concept of implicit hydrogen atoms, most properties, except those which were explicitly declared to be sensitive also to this operation, will remain valid.

The library was designed to err on the side of caution with respect to the validity of data. Maintaining a consistent data set under all circumstances is a high priority. In case an structure modification operation would result in loss of data which should not happen, it is possible to lock existing property data, or even all data on a specific subobject type on a major object. If locking is active, none of the normal data consistency checks apply to the locked data.

In some cases, locking is inevitable because changing of property data can have cascading irrevocable effects. For example, changing data in the property `A_SEARCHINFO` will invalidate the element symbols in `A_SYMBOL`, because some pseudo atoms have specific symbol representations. Invalidating `A_SYMBOL` however will invalidate `A_ELEMENT`, and now all atom element information has been lost. In such cases, an essential property such as `A_ELEMENT` should be requested and then locked until the potentially cascading operation has been completed. For normal structure operations the built-in consistency manager is a clear benefit, not a nuisance.

Property Naming

Property names in the Cactvs system follow a logical scheme.

- The first character is a prefix, which identifies the object class instances of property data are attached to.
  Example: `E_NAME` is an ensemble property.
- The prefix is separated by an underscore from the remainder of the name.
- The following sequence of letters, digits and underscores is the body of the name. Only upper-case letters are allowed, and the underscore is the only acceptable punctuation character.
- If the body of the property name is enclosed by a pair of asterisks, this is a synthetic property. Synthetic properties are automatically generated under certain circumstances when data needs to be captured, but no descriptor record for the property can be found. This happens for example when reading SD-files with data fields which do not correspond to standard **CACTVS** properties. A synthetic property has only minimal set of information. In most cases, not even the data type can be established reliably, so the data is stored in neutral form as strings.
  Example: `E_*SDFIELD1*` is a synthetic property.

- The next section which may be present is a pound/hash character, followed by a number. This number is a property ID. Property IDs are a tool to avoid naming collisions. Property IDs may not be generated by normal users defining their own properties. Rather, the idea is to establish a central repository where properties are registered and assigned an unique ID. This ID can be used to break collisions between properties with the same name.
  Example: `E_IDENT#1` identifies the built-in system property `E_IDENT` and no other property of the same name.
- If the name of the property is followed by a percent character, this is a backup property. Some property definitions contain flags which instruct the system to make a backup of the property before it is changed. An example for such a a property are atom labels (`A_LABEL`). When two ensembles are merged, which often implies atom renumbering of the atoms of the second ensemble in order to avoid collisions, the old labels of the second ensembles are saved in property `A_LABEL%`. Backup properties are identical in all respects to the original property, but they are by definition not computable, even if their originator property is.
- If the name of a property is followed by a slash and a number, this is multiple-instance property data. The Cactvs system allows the storage of multiple sets of data of the same property on any major object. The first instance of a property data may be identified by the suffix /1 (as in `E_NAME/1`), but this is optional (a plain `E_NAME` selects precisely the same instance). The number suffixes may be freely allocated. It is not required to have a sequence without holes.
  Example: `E_IRSPECTRUM/3` selects the third IR spectrum.
- Some types of properties may be indexed. A property subfield is selected by attaching a bracketed index identifier.
  Examples: `E_IRSPECTRUM(INSTRUMENT), E_SCREEN(0)`
  An index may either be the internal field index, beginning with 0, or, if the fields have been named, the field name. Field names must be specified in lower-case. All vector types and compound types are indexible, but indexing capabilities are also provided by pseudo-vectors (float pairs, etc.), string (word indexing), bit sets (bit indexing), blobs (single byte access) and other data types.
- Starting with release 3.358, compound and compound vector properties which have fields that are properties themselves may be accessed in a dot notation. Examples:

  ```
  vertex get $nh $v V_ONTOLOGY_TERM
  vertex get $nh $v V_ONTOLOGY_TERM(synonyms)
  vertex get $nh $v V_ONTOLOGY_TERM(synonyms.size)
  vertex get $nh $v V_ONTOLOGY_TERM(synonyms.7)
  vertex get $nh $v V_ONTOLOGY_TERM(synonyms.7.text)
  vertex get $nh $v V_ONTOLOGY_TERM(synonyms.7.dbxrefs.0.db)
  ```

  Individual field elements of compound vectors may be accessed by a *element.field* notation. This syntax can be used as part of a longer access chain, as shown above. Currently, the maximum number of dot addressing elements that can be used in this syntax is five, the same as shown in the last example.
- Do not use the colon character with a numeric identifier in a property name, such as `E_NAME:2`. This feature is reserved for future selection between multiple computation functions associated with a single property definition and not yet officially supported.

All these property name elements may be combined, *in the order listed*.

Example:

`E_*SOMEDATA*/3(5)` will select the sixth word (index begins with 0, synthetic properties are strings) from the third instance of the data associated with property `E_*SOMEDATA*`.

Various schemes exists which associate additional names with a property.

All property definitions have an attribute “original name”, which may be set to any string. In output operations, **CACTVS** will generally use this name, if it is set, in preference to the internal system property name. The original name is also used for property identification in input operations. If the name of a data field in an input file matches an original name of a property description, the field data is stored as data of the type and name identified by the matched descriptor record. In many contexts (but not all) an original name may be used as a completely equivalent substitute to the system name when referring to properties and property data. Note however, because there is no control on the used character set, original names cannot be used for indexing, multiple-instance data and other operations which require the parsing of the name into subfields.

Example:

```
prop set E_IDENT origname Company_ID
```

sets the original name of the built-in standard property `E_IDENT` to “Company_ID”. If an SD file is read afterwards which contains a data field “Company_ID”, its data will be stored as property data `E_IDENT` on the read ensemble.

Properties may also be *aliased*. A property alias is a redirection mechanism which tells the system that, whenever a specific property name is used, it is substituted by another name. This mechanism allows the convenient mapping of multiple property names as they might be occurring during a processing sequence onto a common internal property. Redirections may happen in multiple steps.

Example:

```
prop alias E_IDENT E_NAME
```

If this alias is set up, `E_IDENT` will only be another name for `E_NAME`, hiding the original definition of `E_IDENT` completely. The alias name however does not need to be an existing property name. It may be any name which follows the property naming syntax. Aliases may be layered, i.e. an alias may refer to another alias instead of an original property name, but the alias structure will contain a reference to the original property and not to any intermediate alias name because aliases to the property name are resolved when the command is executed.

Aliases can be removed with a command like

```
prop unalias E_IDENT
```

where the arguments are one or more alias names, not the original property names the aliases are referring to (`E_NAME` in this example).

Some property value types do not allow the full range of field indexing for setting. Currently, the predefined set of fields in property value types `P_URL` (such as *hostname*, *port*, *protocol*, etc.) and `P_DATE` (such as *year*, *hour*, *weekday*) may only the read, but not set individually.

Object Identification

The standard methods for identifying objects have already been presented:

- Major objects are identified by their handle
  Example: **`ens get ens0 E_NAME`**
- Minor objects are identified by a combination of a major object handle and an object label
  Example: **`atom get ens0 1 A_ELEMENT`**

However, the scripting language provides a number of additional methods for minor object identification:

- Identification via backup labels. If a percent character is appended to the label, the label is interpreted as referring to the backup label property. If backup label property data does not exist, an error results.
  Example: **`atom get ens0 1% A_ELEMENT`**
- Identification via object list index. Prefixing a number if the pound/hash character implies that the object is the nth object in the object list. The first list element has index 0.
  Example: **`atom get ens0 #0 A_ELEMENT`**
- Identification via property value comparison. Any property of the same object class association type as the decoded object may be used in an equality comparison operation. The label argument must be a list consisting of the property used for the comparison and the property value. Note that in case a property value occurs more than once, only the first minor object with the property value is found. The comparison value must be provided in a format which can be decoded to the data type of the comparison property.
  Example: **`atom get ens0 {A_LABEL 1} A_ELEMENT`**
- Bond, and only these, may also be identified by a list of the atoms participating in them.
  Example: **`bond get ens0 {1 2} B_ORDER`** gets the bond order of the bond between atoms 1 and 2. If no such bond exists, an error results.

The property definitions of the minor object label properties (`A_LABEL`, `B_LABEL`, etc.) guarantee that these properties remain valid under almost all circumstances. In case there should be no valid minor object labelling at any time, a default labelling numbering the objects in their list in sequential order, starting with 1, is generated.

Operations which potentially will result in a label renumbering, for example the merging of ensembles, will save the values prior to renumbering as backup properties (`A_LABEL%`, etc.). These backup properties can be used to access data in the new ensemble via saved labels from the old un-merged ensemble.

Whenever labels are generated or minor objects are combined, the system will take care that the primary minor object label values are without collisions.

Filters

Filters are objects which operate on chemical objects, but are not chemical objects themselves. They are often used to select specific objects from larger object collections, for example atoms with specific properties from the atom set of an ensemble.

Using filters for this kind of selection operation is efficient. Filters have significantly less computational overhead than script commands which retrieve property data and compare it to some values.

The **CACTVS** system provides a convenient set of built-in filters. Additional filters can be created on the fly, or existing filters be modified, by the **`filter`** command.

Filters are referred to by their name. By convention, filter names are simple, self-explanatory lower case strings. However, this is not enforced - filter names may contain blanks and use arbitrary characters.

If a name of a filter is used which is not yet loaded into the system, the toolkit will attempt to find a filter definition file in the filter search path. A filter definition file is an ASCII file which contains keyword/values pairs describing the internal set-up of the filter. The file name is the name of the filter, with a suffix *.fil.* Filter definition files are usually created by using a filter definition manager, or by generating them by a script and dumping the filter to a file by a script command. If a filter name cannot be resolved, an error is raised.

In its simplest form, a filter consists of a property name, a comparison value, and a comparison operator. For example, the statement

```
filter create carbon property A_ELEMENT value1 6 operator eq
```

defines a simple filter which will check whether the object data for property `A_ELEMENT` is equal to six. This filter, when applied to a set of atoms, will only let carbon atoms pass. If data of the filter property is not yet present on the filtered objects, it will be computed, it this action is not explicitly prohibited by the filter definition. If the computation fails, the command using the filter will fail.

One of the most interesting features of filters, but also a source of complexity, is their ability to operate on objects whose class does not correspond to the object class the filter property is attached to.

Example:

```
ens bonds $ehandle carbon
```

The application of the *carbon* filter defined above to the set of bonds in an ensemble is completely legal. Generally, when there is a mismatch between the filtered object type and the property object type, an expansion step is inserted, replacing the original object by a list of objects of the class of the filter property the original object is contained in or participating in. In this case, every bond is internally represented by the (usually 2) atoms in the bond, and the property values of these atoms will be compared against the filter value.

Since now more than two comparison take place, the question is what will happen if these comparisons yield different results, for example in the case of a bond between a nitrogen atom and a carbon atom. In the default case, whenever an object replacement occurs, the filter will let the original object pass when any single replacement object passes the filter conditions. In the example case, a carbon-hetero bond would pass. However, using special filter configuration options, it is possible to write a filter which will only carbon-carbon bonds pass if such a feature is required.

Filter object expansion is supported for all classes of structure and reaction objects.

More examples:

```
reaction ens $xhandle product
```

Get the product ensemble from a reaction.

```
mol atoms $ehandle $label 6ring
```

Get the labels of those atoms in the indicated molecule which are members of one or more six-membered rings.

Property descriptions may contain filters as part of their property description record. Filters on properties are used to specify a subset of objects the property is defined for - for example, classical atoms only (no superatoms, 3D points in space, etc.), or atoms with valid 3D coordinates.

Filter Sets and Filter Modifiers

Many commands on chemical objects, especially those performing data retrieval operations, support an optional parameter called a *filter set.* In some cases, this parameter may be augmented by another parameter following the filter set called the *filter modifier.*

A *filter set* defines a set of conditions the requested object must meet in order to have their data returned. The *filter modifier* changes the kind of data which is returned and may impose additional restrictions on the selected objects.

Filter Sets

A filter set is a list of filter names. Optionally, the first element of the list may be an integer number. An empty list is equivalent to not providing any filter list at all. Since filter names may contain white space, it is a good idea to quote the names.

By default, the filter list operates in an *and* mode. All conditions must be met to allow further processing for the request on the object. If the first argument in a filter list is a number, this number indicates the number of filters which let the object pass. Using 1 for this number is equivalent to an *or* mode. Setting it to 0 effectively disables filtering.

Examples:

```
atom neighbors $ehandle $label carbon
atom neighbours $ehandle $label carbon
```

retrieves the labels of all carbon atoms which are bonded to the atom.

```
ens get $ehandle A_FORMAL_CHARGE {1 oxygen nitrogen}
```

retrieves the formal charges of all oxygen or nitrogen atoms in the ensemble.

Filters in filter sets may be individually inverted by prefixing their name with an exclamation mark.

Example:

```
ring get $ehandle $r A_LABEL {!carbon doublebond}
```

will find those labels of the atoms in the ring which are not carbon atoms but do participate in a double bond.

Simple Filter Lists

In some contexts, not all of the functionality of filter sets is available. Instead, only a simple list of filter names, optionally negated by a prefixed !, is understood. Selecting the number of filters that must match is not possible. All listed filters must let the filtered object pass.

Simple filter lists do not occur in contexts which allow the use of filter modifiers.

Simple filter lists will be gradually phase out and replaced by more powerful filter sets. This change is compatible because filter sets provide an exact superset of the functionality of filter lists.

Filter Modifiers

A filter modifier is another list, which may be specified as an optional additional parameter after a filter set in some contexts. This list may contain of an arbitrary combination of the keywords *count*, *bool* (or *boolean*), *include* and *exclude*. The *include* and *exclude* keywords must be followed, as a second list element, by a list of minor object labels.

The count modifier will change the behaviour of the command to return only an object count, not the labels or data of the objects which pass the filter.

Example:

```
ens atoms $ehandle carbon count
```

will return the number of carbon atoms in the ensemble. The *bool* modifier is similar to *count*, but will simply return 1 if any object passes the filter set and 0 if not.

The include and exclude lists define a list of objects which are eligible or not eligible for processing. If they are not set, all objects on the major object are processing candidates. An include list specifies objects which will be subjected to the filter test. An exclude list will remove objects identified by the labels therein, but will not block unreserved objects. If both an exclude and include list are specified, only the objects listed in the include list and not the exclude list are processed.

Example;

```
set alist [atom neighbors $ehandle $l1]
atom neighbors $ehandle $l2 {carbon multibond} [list count include $alist]
```

will return the number of neighbors of the atom with the label in `$l2` which are carbon atoms with a multiple bond and are also neighbors of the atom with the label stored in variable `$l1`.

All objects support a *filter* subcommand to check whether it passes a simple filter list.

Example:

```
atom filter $ehandle $label [list carbon !aroatom]
```

will return 0 if the filter set stops the atom, 1 if it lets it passes - which means that it is a non-aromatic carbon atom.

Obtaining object cross references

The Tcl scripting language environment provides an extensive group of commands to obtain the object identifiers of object which are related to the original object. The relationship may be that of a major object managing minor objects, or an intersection of minor object lists.

Examples:

```
ens atoms $ehandle
bond atoms $ehandle 1
mol rings $ehandle 1
vertex connections $nhandle 1
```

The first command will retrieve the labels of all atoms from the managing major object. The second command gets the labels of the atoms forming the bond. The third example retrieves the labels of all rings which are in the first molecule in the ensemble[1]. The final example shows that these cross referencing mechanisms are not limited to structure data - they are also available in networks, which manage vertices and connections as minor objects.

Within the major object groups of ensemble-related data and network-related data, the full matrix of cross-referencing possibilities is implemented. However, one cannot combine object identifiers from those two realms, i.e. **`ens vertices $ehandle`** is not supported and does not make sense since the result list would always be empty.

Self-references are supported, and are actually useful, because of the possibility to identify a minor object not just by its label.

Examples:

```
bond bond $ehandle [list 1 2]
atom atom $ehandle #3
```

Above examples return the bond label of a bond identified by the labels of the bond atoms, and the atom label of the third atom in the atom list of the ensemble.

Self-references for group objects are a special case, because groups are recursive. A group can contain another group. For this reason, the command

```
group group $ehandle 1
```

will, as the other similarly named commands do, return the group label, while with a plural *groups* as in

```
group groups $ehandle 1
```

reports the labels of all groups which are contained in group 1, or an empty list if the group only consists of atoms. Because of the variability of the objects which form a group, the statement

```
group atoms $ehandle 1 count
```

does not return the correct size of the group, if the group contains other groups. In order to access all objects in a group regardless of the object class, groups have a special *objects* subcommand, which returns nested lists consisting of the object class name and the minor object label.

Two restrictions on cross referencing major objects exist. First, there are no commands to obtain the label of the controlling major object from a minor object. A command such as **`atom ens ens0 1`** would be really useless, since the controlling major object handle is already needed to address the minor object. Second, hopping from a minor object to a major object which is not the controlling major object is also currently not implemented. Attempts to obtain the reaction handle from an atom participating in a reaction via a statement like **`atom reaction $ehandle 1`** will currently fail.

It is however possible to obtain major object handles from related major objects via cross referencing. The commands

```
reaction ens $xhandle
```

1. As a special complication, it is possible to have rings which are not just part of a single molecule, if the bond types which are used for ring detection and separation of atoms into molecules are not identical. The default bond sets are identical, but since the sets may be changed, data analysis and computation routines must not assume that this complication cannot occur.

```
ens reaction $ehandle
queue ens $dhandle
dataset ens $thandle
```

and others of this type are all implemented. If a major object is not a part of a referred major object class, such as in case of an ensemble which is not a component of a reaction, these commands will return an empty list. Otherwise, the related major object handles (not the labels, as in cross references to minor objects) will be returned.

From a syntactical perspective, plural forms of the names of retrieved object identifiers must be used if the statement can possibly return more than one result element, and singular forms if only one element can be returned, or alternatively a single element or an empty list. The plural of *ens* is defined to be *ens*, the plural of *ringsys* is *ringsys*, and the plural of *vertex* is *vertices*.

An ensemble can only be a member of a single reaction and/or a single dataset. A reaction can only be a member of a single dataset, but contain multiple ensembles (but the plural does not differ here). Any minor object can only be controlled by a single major object. Rings *can* (under specific circumstances) be a member of more than one molecule, so the correct form is "**`ring mols`**". An atom however can only be a part of a single molecule[2], so the official form is "**`atom mol`**". Bonds of more exotic types which are not used for separating atoms into molecules may span molecules, so it is "**`bond mols`**".

For historical reasons, not all of these syntactic singular/plural rules are strictly enforced. However, code written today should adhere to the syntactic rules because backwards compatibility is not guaranteed indefinitely. It is dangerous to write scripts in a way that they expect a single returned identifier when the returned data could possibly be a list of identifiers with more than one element. The distinction between singular and plural forms helps to shape awareness of this issue.

Atoms and bonds provide an additional *neighbors* subcommand which can be used to obtain information on neighbour objects separated by one or more bonds (spheres) from the requestor object.

The labels returned by minor object cross reference commands correspond to the *x*_LABEL properties[3], where *x* is the property attachment class prefix of the object class. In principle, the code examples

```
ens atoms $ehandle
ens get $ehandle A_LABEL
```

are performing the same operation. The direct cross referencing command is somewhat more efficient.

Besides being shorter and more efficient, the direct cross referencing commands provide extended capabilities for filtering the result set. The standard property retrieval commands only support the use of a filter set, while the cross referencing commands support an additional filter modifier parameter.

Example:

```
ens atoms $ehandle carbon count
```

2. It is however possible that certain pseudo atoms are not part of any molecule.
3. The label property associated with a minor object class is actually configurable, but it is very strongly recommended not to tamper with it.

returns directly the number of carbon atoms in the ensemble. The equivalent expression

```
llength [ens get $ehandle A_LABEL carbon]
```

is significantly slower, because of the property decoding overhead and the required construction of an internal Tcl value list, which is immediately discarded after its length has been determined. Still, this is much better than

```
set cnt 0
foreach a [ens atoms $ehandle] {
   if {[atom get $ehandle $a A_SYMBOL]==”C”} {
      incr cnt
   }
}
```

In some cases, it is useful to know the position of an object in the object list of the controlling major object. All minor objects provide an index subcommand, which returns this information. The first position is index 0.

Example:

```
atom index $ehandle 99
```

The index position is not necessarily the same as the label minus one. Labels may be changed by many commands, directly or indirectly, and they do not need to form any uninterrupted sequence. The only requirement is that within a minor object set under a controlling major object, no two minor objects of the same class have the same label. On the other hand, a minor object label does not change if the object is moved around in its object list.

Computing data for chemical objects

Obtaining data from chemical objects for analysis or export is one of the most important operations of the toolkit.

Generally, the toolkit relies an a lazy computation approach. Date is usually generated only when a specific data item is requested. At this moment, the toolkit will look at what data is already available and the computation function which is associated with the requested property data. If the requested data is already present, the toolkit library will simple return it. If it is not yet present, it will call the computation function, which will itself request input data, which is either directly returned, or generated in a recursive fashion. When the uppermost computation function returns, either the data has been generated, or the whole process has failed and an error is generated. As a side effect, additional data required by the computational functions will often be generated and attached to chemistry objects involved in the computation.[4]

As long as no events take place which invalidate the data, it remains attached to the chemistry object. The property consistency mechanism used to keep all information in a consistent state and the events

4. The standard process to compute property data is fully automatic and does not let the programmer influence the computation path followed. Experimental features involving an exhaustive generation of paths from source data to requested data and the use of rating functions to select preferred paths exist, but are not contained in the standard toolkit distributions. For standard applications, the default computation path is nearly always effective.

which can lead to a discard of previously attached data is described in a separate section of this manual.

In no case the programmer ever has to specify a sequence of explicit function calls to gather data or to prepare input data for high-level functions. These issues are handled in a completely automatic fashion by the underlying toolkit library.

The concept of a computation function is actually rather broad. These are the standard mechanisms used for computation:

- Built-In functions. For the most common properties, the computation functions are built into the core library. Examples are conversion function from element symbol `A_SYMBOL` to period system number `A_ELEMENT` and reverse, computing the free electrons `A_FREE_ELECTRONS` from the number of shell electrons and the bonds an atom is participating in, or getting the molecular weight `M_WEIGHT` by summing up the atomic weights and isotope labelling information on all atoms in a molecule - potentially automatically initiating the grouping of atoms into molecules at this moment if molecule information was not yet present for the ensemble. Built-in functions are written in C.
- Dynamically loaded functions. Computation functions, together with the descriptions of properties they are serving, may (and frequently are) be kept outside the core library. When the computation function needs to be invoked, a shared object or DLL is located using its name in the property description record and the object search path. If the object could be found, and found in a place which passes security constraints, it is loaded and performs the computation just as a built-in function. Dynamically loaded functions are usually written in C, but may also be written in C++ if the main application was linked with a suitable set of run-time libraries. The build environment of the **CACTVS** toolkit contains mechanisms to include modules which are usually shipped as separate modules and property description files into a library. This method, in combination with the capability to compile an application script into a n executable, is used to build stand-alone applications which, even though they use properties, computation functions and other extensions outside the core library, do not require a full toolkit installation for these modules and are completely self-contained.
- Script functions. The computation functions of property descriptions do not need to be written in compiled code. It is possible, and often a very convenient rapid development approach, to code computation functions in Tcl. These functions become an integral part of the property description record and are stored directly in the file containing the description. This, no additional look-up for locating the script code is required. Script functions are executed in a so-called *slave interpreter*. Every script function set associated with a property is executed in its own slave environment. In this environment, the computation scripts may freely create variables etc. without fear of stepping onto data of the main interpreter running the application script. Slave interpreters may be further restricted in their capabilities, for example by disabling file I/O and Internet capabilities, in order to allow the execution of code from not completely trusted sources in a sandbox environment.

- Distributed computation. The computation of any property computation function, regardless whether it is a built-in function, a dynamically loaded object, or a script function, may be offloaded to another toolkit process. In order to use this feature, the second toolkit process is started with flags instructing it to listen to an RPC port for computation requests. The property definitions used by the primary process are changed in such a way that the offloaded functions are linked to a specific host name and port where the computation server is listening. If a request for a distributed computation arrives, the first interpreter will send a packet of credentials and the required information (which is usually not the full amount of data the requesting process has on the chemical object involved) to the computation server. Depending on the set-up, the second computation server will either directly respond with the result, or try to call the requestor process later after the computation is finished in an asynchronous manner. A mechanism to report results at a time when the requesting process is no longer active using a dedicated mailbox drop is also available. RPC-based property computation is currently not supported on Windows.
- Legacy programs. Such programs are easily integrated into the **CACTVS** computation scheme by a script wrapper function. The wrapper function receives the handle of the requesting object as input parameter, extracts the data needed by the legacy program, writes an input file for that program, runs it, extracts the result data and attaches it to the chemical object. Because property interpreters are isolated from the rest of the system and from other such interpreters, they may keep state, for example maintaining an open pipe pair to the external program. By this method, restarting the program for every computation request can often be avoided and efficiency be maintained.
- Another method to integrate legacy applications is the use of an alternative representation adapter. Alternative representation adapters are a special class of modules which copy **CACTVS** structure data directly into a foreign data structure, calls functions, and extracts information from the updated foreign data structure. To use this feature, the external program must be available as a link library, which is linked to the alternative representation module to build a shared object or a DLL. In case the external library performs multiple functions, interfacing to its functions can be streamlined and automatized by the use of a common adapter module.
- A special class of pseudo-computational properties are those which are configured to automatically set the instance data values to the default values of the property when a computation is requested, without actually invoking any function.
- Property computation requests via the SOAP protocol are currently under development.

Retrieving chemical object data

Accessing chemical object data is a very common task when writing **CACTVS** scripts. The scripting language interface provides a common set of access commands which are supported for all chemical objects, both major and minor.

The following access (sub)commands are available:

- **`get`** Get the property data in a Tcl-parseable format. If the data is already present, return it directly. If it is not available, attempt to compute it. If computation fails, return an error. If the property definition specifies enumerated value, and the internal property value corresponds to a symbolic enumeration value, the value is returned as string. Properties of data type *date* will be returned in ISO format (YYYY-MM-DD HH:MM:SS) in readable format.
- **`dget`** Get the property data in a Tcl-parseable format. If the data is already present, return it directly. If it is not available, attempt to compute it. If computation failed, initialize the property data to the default value and return these values.
- **`local`** Get the property data in a Tcl-parseable format, and always re-compute it, just as in the *new* subcommand described below. However, if the computation function supports this, and property data is already present, the re-computation is performed only for the single data item identified by the object descriptor, so that only one property value is updated. For example, a few selected property computation functions support the update of data for single atoms, not just the full atom set in an ensemble, which is the default. If the property computation function does not support local updates, the standard re-computation on the full minor object set is performed. The *local* command is only supported for minor objects, because it is always equivalent to *new* for major objects, which are by definition the only object in their property data object set.
- **`nget`** Get the property data in numerical form in a Tcl-parseable format. If the data is already present, return it directly. If it is not available, attempt to compute it. If computation fails, return an error. Enumerations are ignored, and properties of data type *date* return the value as seconds since 1970. This number is suitable for use with the Tcl **`clock format`** command.
- **`new`** Get the property data in a Tcl-parseable format. Always re-compute it. It computation fails, return an error. Note that the re-computation only discards the requested property, but not any more low-level data present which will be used in the computation. For example, for a *new* request for the ensemble molecular weight `E_WEIGHT`, only `E_WEIGHT` property data will be re-computed, but not the underlying `M_WEIGHT` property data which is used to get the ensemble weight. In order to restart the computation using the atomic weights, `M_WEIGHT` needs to be explicitly discarded.
- **`show`** Get the property data in a Tcl-parseable format. Do not attempt to compute it. If the data is not already valid, raise an error.
- **`sqlget`** Same as *get*, but will return the data formatted with SQL syntax.
- **`sqldget`** Same as *dget*, but will return the data formatted with SQL syntax.
- **`sqlnew`** Same as *new*, but will return the data formatted with SQL syntax.
- **`sqlshow`** Same as *show*, but will return the data formatted with SQL syntax.

If the descriptor record of a requested numerical property contains enumeration information, and the numerical property value within the range of the enumerated set, symbolic names of the stored numeric values are returned and not the internal numerical value.

Example:

```
bond get [ens create CC] [list 1 2] B_TYPE
```

will return the bond type name *normal*, not the integer value 2 which is internally used. In some cases, numerical values are preferable. The *nget* data retrieval command variation will always return the raw numerical data:

```
bond nget [ens create CC] [list 1 2] B_TYPE
```

This command will simply return “2”. There is no corresponding *sqlnget* command variant, because the SQL formatting will always use the numerical values.

Only a small fraction of property computation functions support local updates for single minor objects. Whether the computation function associated with a property supports this functionality or not can be checked with the following code snippet:

```
set does_local [lcontain [prop get $prop flags] localupdate]
```

Examples for functions which do support this feature are the built-in functions associated with the properties `A_LABEL_STEREO`, `A_MAP_STEREO`, `B_LABEL_STEREO`, `B_MAP_STEREO` and `B_FLAGS`.

Data retrieval commands

The access commands for major objects will need the object handle and the property name as identifier, while access command for minor object use the standard combination of major object handle and minor object label.

Examples:

```
ens get $ehandle E_NAME
atom show $ehandle $label A_SYMBOL
```

The retrieval of property data via an object which is not in the same class as the one the property data is associated with is fully supported and in many cases an elegant solution for a variety of problems. The rules for property access via non-matching object classes are the same as for cross-referencing objects.

Examples:

```
ens get $ehandle A_SYMBOL
bond get $ehandle $label A_SYMBOL
mol get $ehandle $label R_SIZE
```

These statements will retrieve the element symbols of all atoms in the ensemble, the symbols of the atoms participating in the bond, and the sizes of all rings which are contained in the molecule.

Retrieving lists of property data items is generally more efficient than individual requests. So, instead of writing a loop like

```
foreach a [ens atoms $ehandle] {
   set sym [atom get $ehandle $a A_SYMBOL]
   ...
}
```

this loop is preferable:

```
foreach sym [ens get $ehandle A_SYMBOL] {
   ...
}
```

In cases where both the label and the symbol is needed, it is worthwhile to remember that the Tcl *foreach* loop instruction supports the walking of multiple lists in parallel:

```
foreach a [ens atoms $ehandle] sym [ens get $ehandle A_SYMBOL] {
   ...
}
```

Identical to the mechanisms used in object cross referencing, the returned objects may be filtered by a filter set. However, unlike object cross referencing, the data retrieval statements do *not* support the second optional filter modifier argument.

Examples:

```
set msizelist [ens get $ehandle M_NATOMS heterocycle]
set rsizelist [mol get $ehandle $mlabel R_SIZE heterocycle]
```

The first example will return the number of atoms in all molecules which contain one or more heterocycles. The second statement yields a list of the size of all heterocyclic rings in the selected molecule.

It is possible to retrieve more than one property by a single command. In this case, a nested list is returned.

Example:

```
ens get [ens create C] [list A_SYMBOL A_NEIGHBORS]
{C H H H H} {4 1 1 1 1}
```

Above statement will return a list which has two sublists: The first sublist contains the element symbols, and the second list contains the number of neighbors of that atom.

It is even possible to mix the association classes of retrieved properties:

```
atom get [ens create C] [list A_SYMBOL B_ORDER]
C {1 1 1 1}
```

The returned nested list contains a single element symbol for the selected atom in the first sublist, and the bond orders of all bonds the atom is participating in is returned in the second sublist.

Setting of property computation parameters

As a final parameter after the optional filter list, it is possible to specify a list of name/value pairs for the computation of the requested property. If the data is already present, the specified computation parameters are checked against the saved computation parameters which were used for computing the existing parameters. If any discrepancy is fund, the property data is re-computed with the new parameters. Only specified parameters are checked - parameters which are not mentioned in the requested parameter list and which are present in the saved parameter set will not trigger a re-computation. Implicitly, the parameters which are used in the computation request are also set as default for future computations inside the property definition data structure.

Example:

```
ens get $ehandle E_GIF {} [list height 200 width 200]
```

This line will always return an image with height 200 and width 200. If an image of a different size is attached as `E_GIF` property data to the ensemble, it will be discarded. This command will not change the current global parameter setting of property `E_GIF`. Parameters not listed in the parameter list will be taken from the global settings of `E_GIF`, but any parameter which is set locally in this statement will be active only for the execution of the command.

Note the use of an empty filter set as fifth word in this example. This empty parameter is required to skip the filter set parameter argument position. An empty filter set is equivalent to omitting the filter set altogether.

The parameters which are recognized depend on the property. Names of parameters which are not used by the computation routine for the requested property are ignored. The names of parameters recognized by a property, as well as its current settings, can be obtained by calling

```
prop get E_GIF parameters
prop get E_GIF defaultparams
```

Both commands return a name/value list of parameter names and values, which can be stored in a Tcl array via an *array set* command. The first version returns the current parameter setting, while the second returns the default setting. Individual values can be obtained via commands like

```
prop getparam E_GIF width
prop getparam E_GIF height
```

Property computation parameters can also be set directly on the property definition structure, via commands like

```
prop setparam E_GIF height 200 width 200
```

The *setparam* subcommand of the *prop* command allows the manipulation of individual parameter values, just like the *getparam* subcommand is used for extracting specific parameter values. Using the *get* or *set* commands on the parameter attribute, which is only one of a large number of attributes which form a property definition, will retrieve or set the complete parameter list in one step.

The **`prop set`** and **`prop setparam`** commands change the parameter values of a property globally within the current program. The changes remain active until the program is terminated, or until they are overwritten by additional commands, or a reloading of the property definition. They will not change the parameter settings persistently - when a new interpreter is started, it will use the original parameter settings. The only way of changing property parameters permanently is by editing the corresponding property definition files.

Resetting parameters to the default value is easily done via statements like

```
prop set E_GIF parameters [prop get E_GIF defaultparams]
```

Property metadata

The toolkit remembers the parameters property data was computed with, and numerous other information about the history and origin of data. Some parts of this meta information may also be edited or augmented by script commands.

All commands to work with metadata are used with major objects only. This makes sense since properties are always computed or otherwise set up for all minor objects of a major object in a concerted action.

The generic command to access property metadata is the *metadata* subcommand for major objects:

```
ens metadata $ehandle E_GIF
ens metadata $ehandle E_GIF comments
ens metadata $ehandle E_GIF comments “this is a beautiful picture”
```

Note that this will work with all major object types, such as tables, reactions, or networks, if you use the corresponding object command and object handle. The first example will return a list of keyword/value pairs of all metadata. It can be stored in a Tcl array variable by means of a *array set* command:

```
array set gifparams [ens metadata $ehandle E_GIF]
```

The second example will selectively return the value of the *comments* field in the metadata record. The third line shows how to set a metadata field.

The standard fields for property metadata are:

• parameters	The parameter set used for computation, as a list of keyword/value pairs.
• info	Information about events which happened during computation as a string.
• comments	Free-text comments as a string.
• flags	A standard set of flags indicating status. Usually, this field is *none*, but it can be a list of the values *unreliable* (applicability of computational method questionable, but not a hard error), *remote* (computation underway on remote server), *async* (remote computation in asynchronous mode), *interpolated* (data is not in original grid, but was interpolated from other data points not on the grid), *reagent* (applies to reagent side of a reaction), *product* (applies to product side of a reaction) and *unoptimized* (basic data was successfully computed, but successive optimization/smoothing steps failed).
• unit	This is not the property base unit (which can be obtained via a **`prop get $p unit statement`**), but rather the unit for grid points of multi-dimensional properties. This field is not available for non-grid data. As an example, the base unit of a property might be fractions of electron charges, while the grid unit might be Ångstroms.
• dimensions	The number of dimensions for grid data. Not available for non-grid data.
• xrange, yrange...	The low and high limits of the coordinates for grid data on each axis, in the units defined above. Not available for non-grid data. This parameter is a pair of floating-point numbers, or empty strings if the low/high bounding data is unknown. The coordinate axis names are x,y,z,w,u,v,a,b,c,d,e,f,g,h in this order. Only the coordinates up to the declared number of dimensions are output.

The dimension count and coordinate ranges cannot be changed.

Additional parameter keywords in the *parameters* metadata field may be conveniently set on existing property data with the *setparam* command:

```
ens setparam $ehandle E_GIF mimetype "image/gif"
```

This command will also overwrite existing keywords in the parameter set. This is a convenience function - it could be replaced by first getting the full parameter list, manipulating it, and writing it back as a whole.

There is also a corresponding *getparam* command:

```
ens getparam $ehandle E_GIF width
ens getparam $ehandle E_GIF format
```

The first line of code will selectively return the value of the *width* parameter which was used during the computation of the `E_GIF` property and which reflects the image width[5]. The second example line of code will report the image format, which can actually be *gif*, *png* or various Windows bitmap types. Note that every property has its own parameter set, which can be obtained via a *prop get $p parameters* statement. These examples are not directly transferable to other properties.

The native **Cactvs** file formats store and thus preserve property metadata. Unfortunately, none of the standard structure exchange formats provide similar functionality, so this information is lost when importing or exporting structures and other data objects in non-native formats.

The metadata command may be abbreviated to *meta*. The deprecated alias *propenv* is also still supported.

Indexed access to property data

As it has been explained in the chapter on property naming, many properties allow indexed access to subfields of the property. The precise meaning of indexed access depends on the data type of the property in question. Some examples:

```
ens get $ehandle E_FILE(name)
```

The first example line demonstrates the subfield access to a compound property. `E_FILE` is a property which is automatically added by the structure file input routine. If contains the name of the file the structure was read from, the record number before and after reading the data[6], the line number of the beginning of the read file section, and the file format name. The file name is stored in the field named *name* of the `E_FILE` compound data structure, and may be accessed directly by specifying its name. If the full property data without indexing is returned, a list of all the data fields is returned. The name of the field is set in the property definition record for `E_FILE`. Alternatively, its numerical index, which is 0, could have been used. Using the symbolic name is generally preferable, since it makes the code more readable and the name will not change if fields are added or removed from a later version of the property definition.

```
ens set $ehandle E_NAME "a b c"
ens show $ehandle E_NAME(1)
```

5. This is actually only true if there is no image cropping. The *width* field is the original width of the image used for drawing the structure. The final image height and width are provided in the *croppedwidth* and *croppedheight* fields. If cropping is disabled, these are the same as the original width and height values.
6. Under certain circumstances, it is possible that an input operation consumes more than one file record. This is the reason why two record numbers are stored in these property data instances. For standard applications, both record numbers have the same value.

The second indexing example shows the use of a numeric index on a string property. This example will return the second word (the index positions begin with 0), which is "b".

Finally, the statement

```
ens get $ehandle A_XY(x)
```

will only return the X-coordinates of the 2D display coordinates of the compound.

Property computation requests without data retrieval

In some cases, a script is not interested in accessing the data items, but just wants to make sure that certain property data is present. This can be achieved by the *need* subcommand:

```
ens need $ehandle A_LABEL_STEREO
```

This command will not return the actual data values, but start a computation if the data is not yet present. If the computation fails, an error is raised. It is possible to use a property name list instead of a single property name.

Optionally, a processing flag list and a computation parameter list may be specified as well:

```
ens need $ehandle E_GIF recalc {width 200 height 200 bgcolor white}
```

The mode flags argument may be empty, or any combination of the flags *recalc* (force recalculation), *reload* (force reloading of computation module if it is an external module), *default* (no computation, just set to default, but preserve if already present), *reset* (make sure that property is attached, and reset to the default value), *ifcomputable* (compute if possible, otherwise set to default), *defaultonerror* (if computation raises error, reset to default but ignore error), plus a couple of undocumented specialist options mainly intended for debugging.

Checking data presence and applicability

The *valid* subcommand is available for major objects to check whether a property is valid for that object.

Example:

```
ens valid $ehandle A_ISOTOPE
ens valid $ehandle E_NAME
expr {![catch {ens show $ehandle E_NAME}]}
```

the first two code lines will return 1 if the properties `A_ISOTOPE` or `E_NAME` are part of the data attached to the ensemble, 0 otherwise. Note that this query always operates on the controlling major object. This is due to the fact that either all minor objects under a major object have a valid property value, or none has, so that an individual check on a minor object does not make sense. The third code line performs the same operation as the second one in a convoluted way - the *show* subcommand will raise an error if it is used on non-existing data.

However, having a valid property as part of the major object data does not necessarily mean that the property is defined for an individual object. It may well be that, for example because an atom is of an exotic pseudo-atom type, some property data for such pseudo atoms has no meaning. Usually, the property value for these objects will be the default value of the property, or be set to a magic number, but this is not reliable. The simplest way to check whether a property is actually meaningful for an object is to use the *defined* subcommand.

Example:

```
atom defined $ehandle $label A_XYZ
```

will return 1 or 0, depending on whether the concept of 3D coordinates is defined for that class of atom, or not. Limitations to the applicability constraints of properties are implemented as a filter set which is part of the property definition. An empty filter set does not impose any applicability constraints.

Currently, the property management mechanism do not support explicit **`NULL`** values on chemistry objects. Table element data is an exception - tables already support the notion of unset data on rows, columns and cells, but not in table-global property data. This feature will be added in generic form in the near future.

Setting property data

The scripting language interface provides three subcommands for setting data on chemical objects. These subcommands are available for all major and minor chemical objects.

- set — Create new property data information. If the data is already present, it is overwritten.
- append — Append property data information. If the data is not yet present, it will be computed and the new value appended to the computed data.
- fill — Essentially the same as the set subcommand, but the value list may be shorter. The missing elements are initialized to the default value of the property.

The precise meaning of "appending" data depends on the data type of the property values which are manipulated. The following rules apply:

- For vector data types, the new data is appended by adding elements to the vector.
- For string data, the new data is appended by concatenating it to the current value.
- For numerical data, both integer and floating point types including hashcodes, which are 64-bit unsigned integers, the new value is numerically added to the old value.
- For bit sets, the bits in the new data are bit-ored to the existing value.
- For tree-type data (data and query trees), the new value is linked as a new child of the topmost tree node. If the old tree did not have any nodes, the new data becomes the tree.
- For other data types, *append* is treated as *set* and will just replace the old value.

The rules are applied in this order, meaning that for example float vector data will be extended by adding new vector elements, not by adding the two vectors. Other behaviours can easily be enforced by requesting the data item with a *get* command, manipulating it by any method, and then overwriting it with a normal *set* command.

Examples:

```
ens set $ehandle E_NAME "New "
ens append $ehandle E_NAME "lead"
```

After these two commands, the ensemble name will be "*New lead*".

It is possible to set or append to indexed fields of property data selectively without changing the rest of the data.

Example:

```
ens set $ehandle E_NMRSPECTRUM(instrument) "Bruker PaceMkrStopper 2003 Ultra"
```

More than a single property can be set in one command. After the object has been identified, an arbitrary number of property and data pairs may be used as arguments.

Example:

```
atom set $ehandle $label A_COLOR red A_FLAGS boxed
```

The associated property class of the modified property does not have to correspond to the class of the manipulated object. If there is a discrepancy, the same object replacement algorithm as in object cross referencing and object data retrieval is invoked. If the *set* or *append* methods are used, the number of values passed in must correspond to the number of objects after substitution. In case of the fill method, superfluous data items are ignored, and missing data items substituted by the property default value.

Example:

```
bond set $ehandle $label A_COLOR [list red red]
```

This command sets the colour of the two atoms which form the bond (assuming it is a standard bond with two atoms) to *red*. Using more than two colour data items, or less, will result in an error.

Property data consistency manager issues

The toolkit has an automatic mechanism to keep the overall property data set in a consistent state. Property definitions contain information about underlying, more basic properties the values depend on. When property data is changed, all other properties which rely on the changed value will be purged. The process is recursive, and all properties whose data was invalidated in the first generation will be submitted to another round of dependency checking. Changes on core data such as the element number will result in the loss of most ensemble information.

In some cases, this mechanism can overshoot. A common example is setting of `A_SEARCHINFO` atomic query data. Example:

```
atom set $ehandle $label A_SEARCHINFO(ringcount) [expr ~1]
```

This line sets the match condition that the atom must be a member of one or more rings. However, the molecular ensemble will afterwards most likely become almost unusable. The reason is that `A_SYMBOL`, the atomic symbol, depends on `A_SEARCHINFO`, because special symbols such as *?* for an *any* atom, or *L* for an element list are produced when `A_SYMBOL` is computed from basic information, which includes the properties `A_TYPE`, `A_ELEMENT`, and `A_SEARCHINFO`. Now, when `A_SYMBOL` is invalidated, `A_ELEMENT` is also in the next round of dependency checking, because `A_ELEMENT` is computed from `A_SYMBOL` when it is not present. Thus, we end up with neither `A_ELEMENT` nor `A_SYMBOL` and have lost all element information and there is no way to re-compute the information. The next time element information is needed, an error message about possible infinite recursion will be generated, because the toolkit is caught in a loop trying to get `A_ELEMENT` from `A_SYMBOL`, and `A_SYMBOL` again from `A_ELEMENT`.

The way to prevent this from happening is to lock the element information:

```
ens lock [ens need $ehandle A_ELEMENT] A_ELEMENT
atom set $ehandle $label A_SEARCHINFO(ringcount) [expr ~1]
ens unlock $ehandle A_ELEMENT
```

The first line makes sure that `A_ELEMENT` is present, and then locks it, making it insensitive to the normal dependency checks. After setting the query data, the element information is unlocked again and put back under the control of the data manager.

Fortunately, in most cases the property data consistency manager does not interfere in unexpected ways and setting property data is straightforward.

Object Attributes

Not all data attached to chemical objects is stored as property data. Internal state information of the objects are accessible as attributes, not as property data.

An attribute can be distinguished from a property by the name. Object attributes are always simple, lower case, single words. These cannot be confused with property names in **CACTVS** nomenclature, since these are always written in uppercase. Attributes cannot be indexed. In case of a collision with an alternative property name (such as a data field name from an SD file, etc.), the object attribute has precedence in the identification process. The attribute set for each object class, which is directly linked to the object data structure, is fixed and cannot be changed without recompilation. There are no definition records or other meta-level description mechanisms for attributes.

The number of attributes associated with a chemical object is highly dependent on its object class. Structure file objects have dozens of attributes, but usually no or very few properties. Ensemble objects typically store a rich set of property data, but have only very few attributes. Most minor objects possess no attributes at all, since their state is generally managed by their major object and they cannot exist without it.

The object attribute access commands are the same as for object properties. All retrieval commands for attributes return the attribute value in Tcl format, without any mechanisms to change the formatting. Most attributes only support a simple *set* operation. The only exception are bit sets, which also provide an *append* method to allow the addition of specific attribute bits. Attributes may only be queried and manipulated directly on the current object. Implicit or explicit cross-referencing to other objects is not supported. Many attributes are read-only, but not necessarily constant. In some cases, changing the value of an object attribute will result in major internal reorganization, which can have far-reaching side effects.

A few selected object attributes are saved and restored if native **CACTVS** storage formats are used, but not all. For example, the ensemble modification count attribute is not saved. It is reset to zero when an ensemble is created, and is already incremented during the initial input process, regardless of the file format read. Traditional chemical information exchange formats will not preserve any object attributes.

Examples:

```
set fmt [molfile get $fhandle format]
molfile set $fhandle format sdf eoltype unix
molfile append $fhandle readflags noimplicith
```

```
ens get $ehandle modcount
```

Atoms and Bonds

The **CACTVS** library has a very broad view of what *atoms* and *bonds* are or could be. Scripts should be written to be prepared to cope with unexpected atom and bond types.

The atom and bond types are declared by values of the properties `A_TYPE` and `B_TYPE`. These are essential properties and should never be deleted. Both properties are encoded as bit positions on a 32-bit integer. All acceptable values of the type properties are powers of two. A type declaration has only a single bit set, but in other contexts multiple bits may be set to form a mask of acceptable or unacceptable types. In principle, it is possible to declare new atom and bond types at runtime.

The type property only defines the general class of the atom or bond. For example, an atom with `A_TYPE` *normal* will be fully defined only by additional data in properties `A_ELEMENT` and/or `A_SYMBOL`, possibly `A_ISOTOPE`, `A_FORMAL_CHARGE` and/or `A_FREE_ELECTRONS` plus bonding information, and so forth.

Atom types

These types of atoms are supported in the standard toolkit distribution:

• *normal (1)*	A standard atom with an element number. This is what is usually considered an atom. It is the only type of atom where electron counting is performed for bonds.
• *search* (2)	A search specification, such as an *any* atom, or a list of possible elements. The query data is stored in property `A_SEARCHINFO`.
• *epair* (4)	An electron pair. This type of pseudo atom is usually generated as result of reading certain modelling software files. The **CACTVS** toolkit does not natively encode electron pairs as pseudo atoms.
• *3dpoint* (8)	A point in 3D space without a nucleus. Examples are grid points with property data such as NMR shielding, or a point in space used as reference in 3D substructure searching.
• *super* (16)	A superatom, which is a placeholder for a larger group of atoms. Known superatoms may be expanded.
• *delocanchor* (32)	A 2D or 3D point related to a delocalized π system. An example where this is used are charge symbols placed in the vicinity and connected to a π system with a delocalized charge for depiction purposes.
• *polymer* (64)	A generic polymer, for example a bead in solid phase synthesis. The standard depiction methods in the toolkit allow the display of these pseudo-atoms as a bead.
• *annotation* (128)	A generic placeholder for non-structural annotations to a 2D or 3D depiction, such as comments, arrows, etc.

• *open* (256)	A generic placeholder for an open valence. These are generated by reading certain data sources - **CACTVS** does not encode open valences by itself as pseudo atoms, but as atom attributes.
• *enzyme* (512)	A generic placeholder for an enzyme or peptide. This type has been used for the modelling of metabolic pathways.

From this list, only the types *normal*, *search*, *3dpoint*, *super* and *polymer* are routinely encountered. Some of the more exotic types were introduced only for specific projects and should not be considered fully supported in all contexts.

Bond types

This is the list of standard bond types:

• *link* (1)	A neutral indication for a relationship between the involved atoms.
• *normal* (2)	A standard valence bond. This is the most common type of bond. The electrons needed for the bond are automatically subtracted from the `A_FREE_ELECTRONS` count of the involved atoms (if these atoms are normal atoms) when the bond is formed. Likewise, the standard commands for bond manipulation will update the atom electron counts when changing the bond order `B_ORDER`, or cutting the bond.
• *hydrogen* (4)	A hydrogen bond (*not* a bond to a hydrogen atom - these are encoded as normal VB bonds).
• *dative* (8)	A dative bond. This bond type is treated slightly differently from the more common *complex* bond type. For example, it is not contained in the set of bonds which are used to define molecules.
• *3center* (16)	A three-centre bond, as they are for example found in boranes. Note that this bond contains three atoms!
• *angle* (32)	A pseudo-bond used to encode bond angle information. This bond contains three atoms. The middle atom is the atom for which the angle to the other two atoms is computed. It it not required that normal bonds between the first and middle, or the middle and third atom exist, so this construct can be used to measure the angle of any atom triangle.
• *torsion* (64)	This pseudo bond is very similar to the *angle* pseudo bond. It is used to store torsion angle (the angle between the plane formed by the first three atoms and the plane formed by the last three atoms) information. This bond contains four atoms. It is not required that there are any normal bonds between the atoms which form this pseudo bond.
• *rgroup* (128)	This bond links one or more alternative R-groups to a core fragment. It is for example used in substructure searching where different substituent groups are possible. The first atom in the bond is part of the core fragment, all other atoms are the link atoms of alternative fragments.
• *nobond* (256)	This bond type is used in substructure searches. It declares a bond which must *not* be present in a matched structure.

• *deloc* (512)	This uncommon pseudo bond type is used for the encoding of delocalized systems. It is generated by reading data from certain data sources like the *SpecInfo* database which use this bond type for the encoding of delocalized charge or tautomer systems. This bond type, which was introduced for a specific project, is not fully supported in all contexts and should be avoided.
• *complex* (1024)	A generic complex bond. It is the preferred method to encode metal complexes. In most respects, it behaves like a normal VB bond, but electrons for bonding are not counted or consumed. The standard **CACTVS** display modules will depict these bonds as dotted lines, without a need for setting this attribute bit in `B_FLAGS` explicitly.

The most common bond types are *normal* and *complex*. The file I/O routines of the toolkit contain routines to convert almost any compound into a reasonable representation with a mixture of normal and complex bonds.

Bond class sets

The sets of bonds which are used to sort atoms into molecules, and to find rings in ensembles, are configurable as a bit mask, for example via the global control array elements ::**`cactvs(molbond)`** and ::**`cactvs(ringbond)`**. The standard set of molecule-defining bonds consists of *normal*, *3center* and *complex*. This is also the standard set of ring bonds. Note that these bond sets are indeed independent - under the right circumstances, it is possible to have rings which span multiple molecules.

Another important bond set is the set of persistent bonds, which may be modified via the control variable **`::cactvs(persistbond)`**. This set contains by default the bond types *normal*, *nobond*, *3center*, *rgroup* and *complex*. Only bonds of these types survive any modification of the bond list. Pseudo bonds such as torsions and angles will automatically disappear whenever the bond list is edited.

Aromatic bonds

There is no aromatic bond type in the **CACTVS** toolkit. Internally, aromatic systems are managed as a Kekulé structure with single and double bonds. However, **CACTVS** is of course aware of bond aromaticity. This information is encoded as additional properties `B_ISAROMATIC` (a boolean flag) and `B_ARORING_COUNT` (integer, number of aromatic rings the bond is a member of). The toolkit contains a aromatic system resolver, which will automatically generate a Kekulé form from data sources which encode aromatic bonds explicitly as such. The original aromaticity information is preserved as `B_ISAROMATIC` property data. In the other hand, aromaticity will be detected if the properties are not yet present, so simply using `B_ISAROMATIC` data in any context will implicitly trigger a ring system and aromaticity analysis if this has not yet been done.

One common problem encountered upon reading data from MDL Molfiles is a misinterpretation in the proper encoding of aromatic bond information in these files. The MDL bond type 4 is, according to the official MDL documentation, a *query bond type*, and properly read as such by the toolkit. A normal single bond will be generated, and an addition an attribute flag in `B_SEARCHINFO` is set to make sure that this bond will only match an aromatic bond in a substructure match. However, no Kekulé structure is generated, and such structures appear to have all single bonds. Structure data in Molfiles with aromatic bonds should be written in Kekulé form by conforming programs generating the output. In case aromatic bonds were nevertheless mistakenly output as type 4, the resolver must be invoked explicitly:

```
set fh [molfile open sloppy.sdf r aroresolver 1]
set eh [molfile read $fh]
```

The aromaticity detector of **CACTVS** generally works well, but does not in all cases have an identical opinion as other structure processing toolkits on the question whether a given system is aromatic or not. As an example, the Daylight toolkit has a broader idea on what aromatic systems are than **CACTVS**. For **CACTVS**, rings with exocyclic keto groups cannot be aromatic because the π system is not cyclic, but for Daylight apparently every ring where all member atoms participate in π systems is automatically considered aromatic. For substructure matching, a special mode has been implemented to allow to compensate for this difference in the perception of aromaticity, but in other contexts developers need to be aware of potential differences.

Query bonds

While there is an explicit atom type *search*, no explicit query bond type exists, with the exception of the special case of the *nobond* bond. A bond always has a primary type, which usually is a standard single VB bond, or a *link* bond in case problems with electron counting need to be avoided. Bond query information in then stored in property `B_SEARCHINFO`. Fields in that property will, for example, allow overriding of the actual bond order in the substructure by a list of acceptable bond orders and bond types in the matched structure. If no such attribute is set, the actual bond order must match the bond order in the structure, with special consideration given to aromatic systems where bond orders are not compared directly.

Bond set-up

Under most circumstances, bonds will be present when an ensemble is generated, and they remain present as minor object set throughout the lifetime of the ensemble.

However, this is not an absolute requirement. The minor object group of bonds may be absent from an ensemble, just like rings or molecule, and the toolkit has a built-in mechanism to set up bonds as a minor object group whenever they are needed.

If bonds are required but not present, an attempt will be made to generate them by an analysis of atomic 3D coordinates. The degree of success for this automatic 3D structure analysis varies. I is beneficial to have structures with a full set of hydrogen atoms.

Whenever files which do not contain bonding information are read (such as .xyz files, or some PDB files), the bond generation algorithm is automatically invoked.

Because of the possibility to use pseudo bonds such as torsional angles or bond angles, the bond set of a structure may change even without any structural modifications. Applications should always filter the bond sets they are working on by the bond and atoms types (properties `B_TYPE` and `A_TYPE`).

Ensemble Minor Objects

Besides atoms and bonds, which form the backbone of an ensemble, ensembles may control an open set of additional minor objects. In principle this set is designed to be extensible, but currently no scripting language interface to minor object class extensions exists.

When an ensemble is input, directly as an ensemble or indirectly as a part of a larger structure such as a reaction, dataset, or as attached property data, in almost all cases the standard minor object sets of atoms and bonds is set up.

Automatic initialization

Other object classes are usually not initialized. However, these object sets are set up whenever property data for that object class is requested, either directly or indirectly by some recursive computation function. For example, atoms are often not sorted into molecules. The first time a molecular property is accessed, for example by requesting `M_WEIGHT`, or executing an *ens mols* command which implicitly uses `M_LABEL`, the toolkit will perform assign the atoms to molecules, and set up molecule minor objects, one for each molecule found. These molecule objects then act as anchors for the attachment of molecule properties, such as `M_LABEL` or `M_WEIGHT`.

The same system is used for rings, ring systems, π systems, σ systems, etc. Not all minor object types have set-up functions which actually generate objects. For example, the set-up functions for groups will initialize the group control structure, but not actually generate any default set of groups because there is no standard procedure to generate an initial set of groups.

Loss of minor object sets

In contrast to atoms and bonds, which are very stable with regard to structural modifications and are rarely completely discarded, this is generally not true for other minor object sets. For example, molecule, ring and ringsystem information is completely discarded whenever a bond is made or broken, without any deep analysis whether this operation results in an actual change of molecule or ring information. Rings, ringsystems and molecules are regenerated in a lazy fashion whenever their presence is required the next time. Some specialized operations which are technically speaking atom and bond changes will however preserve additional information - for example, hydrogen addition via *hadd* commands will not destroy ring information because these commands will never change the ring set.

Not all minor object types are completely destroyed as a result of atom and bond changes. For example, groups are more robust - when an atom is deleted which is a member of a group, that group (and potentially other groups which contain the group as elements) is deleted, but not the complete set of groups. The behaviour of a minor object class depends on the class-specific handler function.

Locking of minor object sets

It is possible to lock minor object groups, and thus make them insensitive to structural changes. In case a minor object whose list is locked stores object references, and a referenced object is deleted, the minor object containing the object is deleted instead of the full minor object list. This is a recursive procedure. Under certain conditions, this will work as expected. For example, it is simple and convenient to lock the ring information if the programmer knows for certain that all structure manipulations which are performed during the lock will not change the ring set for the structure. If however an atom is deleted which *is* a ring member, the rings containing the deleted atom are themselves deleted, resulting in an incomplete set of rings. Since the set of rings is still considered to be set up, rediscovery of the ring set will not automatically happen and must be initiated manually - and this requires that the developer is aware of what happened. For this reason, locking of minor object sets should not be considered a routine procedure.

Properties set as result of automatic minor object set-up

The discovery process of minor object sets will set a number of properties on these objects, and potentially on other types minor objects of the controlling major objects. In all cases, the label property for the identification of the minor object is initialized.

All label properties, when initially set, number the objects controlled by the major object in a sequence starting with one. After that, the object label is never automatically changed as long as the object is in existence. Very few exceptions exist, such as the merging of ensembles and similar operations, where minor object labels may be shifted with a constant offset in order to avoid collisions. Because the object label is the primary access key to the object within the scripting language environment, extra care should be taken to avoid collisions when setting them directly.

Object label properties are configured to create automatic back-ups when they are changed. The previous label set is preserved under the original name with a % character suffix, for example `A_LABEL%`.

This is the list of properties set when a minor object list is set up:

• Bonds	`B_LABEL` (bond label), `B_TYPE` (bond type), `B_ORDER` (bond order)
• Molecules	`M_LABEL` (molecule label), `E_NMOLECULES` (molecule count), `A_MOL_NUMBER` (atom molecule index plus 1, 0 for atoms outside molecule[7])
• Rings	`R_LABEL` (ring label), `R_TYPE` (ring class)
• Ringsystems	`Y_LABEL` (ring system label), `R_SYSTEM` (ring system label)
• π Systems	`P_LABEL` (π system label), `P_CLASS` (π system class)
• σ Systems	`S_LABEL` (σ system label)
• Groups	`G_LABEL` (group label)
• Vertices	`V_LABEL` (vertex label)
• Connections	`C_LABEL` (connection label)

Molecules and Rings

Molecules, rings and ring systems are standard minor object classes which are automatically maintained by the toolkit. Set-up of these object classes is fully automatic. There are no mechanisms[8] for manually generating object instances for these classes.

7. Warning: `A_MOL_NUMBER` is *not* guaranteed to correspond to the molecule label `M_LABEL`. Use property `A_MOL_LABEL` for this purpose.

Controlling the detection of molecules, rings and bonds

The automatic set-up of molecules and rings can be controlled in a number of fashions. Important mechanisms are:

• molecule bond types	A bit mask of bond types which are used to find atoms which are linked to a common molecule, which can be configured at the scripting language level in the **`cactvs(molbond)`** control array element.
• ring bond types	A similar bit mask of bond types which define rings in the ensemble. It can be configured at the scripting language level via the **`cactvs(ringbond)`** control array element.
• ring set	The toolkit can be configured to find different kinds of ring sets. The ring set used for automatic ring detection can be changed by means of the **`cactvs(ringset)`** control array element. The most useful values for this field are 0 (SSSR), 1 (extended SSSR) and 3 (full set). The default is the extended SSSR set. The definition of the extended SSSR is that it contains the SSSR rings, plus all rings with a sequence of three consecutive ring atoms which are not contained in any SSSR ring. For example, under this definition *cubane* will have 6 four-membered rings (5 in the SSSR), *anthracene* 3 six-membered rings and two 10-membered rings, but no 14-membered outer ring (in the SSSR, *anthracene* has only three six-membered rings), and *norbornane* two five-membered rings and one six-membered ring (two five-membered rings in the SSSR).

The determination of ring systems do not directly rely on bonding information. Rather, any rings from the current ring set which contain at least two common atoms are considered to the part of the same ring system. Spiro ring pairs are *not* in the same ring system, if no other link than connecting one spiro atom exists.

Ring sets

The choice of the ring set will influence a number of other property values, for example all ring membership counts. In case of metal complexes, the choice of the ring bond types will also have a pronounced effect on the rings detected in the structures. The class of a ring can be queried via the `R_TYPE` property, which is automatically set during the ring detection process.

Example:

```
foreach r [ens rings $ehandle !envelope] { }
```

This code fragment will do something with the rings, but filter out all envelope rings. An envelope ring is a ring which can be constructed by the union of smaller rings. In *naphthalene*, all rings except the six-membered rings are envelope rings. In *norbornane*, the six-membered base ring is not an envelope, because it does not contain the bridge atom present the five-membered rings and is thus

8. There still is a method to create a ring system manually, but this is deprecated functionality.

not a union. The *envelope* filter is a built-in filter which checks the value of property `R_TYPE` to be *envelope* (and not *esssr*, or *sssr*).

The ESSSR ring set has the advantage that it is much more stable with respect to atom numbering than the SSSR[9]. This is for example important in substructure searching with ring membership counts or checks on ring types. For example, a check whether an atom is a member of a heterocycle can easily fail for cage compounds if the heterocycle was by chance assigned to be one of the implicit rings, such as the sixth four-membered ring in *cubane* which is not in the SSSR.

A number of functionalities which rely on ring information were programmed to use only rings of the ESSSR and ignore extra rings which might be present because of a larger ring set was computed. Whether a function or property restricts itself to the use of ESSSR rings, or uses all rings it finds, is documented for the specific functions.

Effects of setting different bond and ring bond type masks

Because the bond types used for the detection of rings and molecules are independent, it is possible to have rings which span more than one molecule, or molecules which contain partial rings. When an ensemble is split into molecules to form individual ensembles, partial minor object structures are lost.

Certain atom types (*undefined*, *3dpoint* values in property `A_TYPE`) are not considered to be part of any molecule, regardless of the bonding situation. These pseudo atoms will have 0 as `A_MOL_NUMBER` or `A_MOL_LABEL` property values.

Molecule manipulations

Molecules support an extended set of commands compared to other minor objects. Most of these reflect the fact that molecules are central to organizing chemistry data and exist as isolatable physical entities.

When molecules are addressed, it generally means that the group of atoms which form the molecule are processed. Information which is encapsulated within the molecule (such atom, bond and ring data) is preserved where possible. Any objects which cross the molecule boundaries (such as rings or bonds which are not covered by the bond types used to define a molecule) will be lost.

Examples:

```
mol dup $ehandle 1
mol delete $ehandle 2
```

The first example will duplicate the molecule with label 1. The molecule forms a new ensemble. Data of the duplicated molecule is preserved where possible, so atom, bond and ring labels will be the same as in the original molecule. Objects which are not restricted to the original molecule, such as molecule-crossing bonds and rings, or groups which contain atoms from different molecules, are not duplicated. Other minor objects of the ensemble, such as other molecules, atoms and bonds outside the deleted molecule, groups which did not contain any atoms of the deleted molecule, etc. are preserved.

9. But is it *not* guaranteed to be completely independent of atom numbering! For practical purposes, this is however usually no an issue, in stark contrast to working with traditional SSSR ring sets.

The second line of sample code deletes a molecule with label 2 from an ensemble. Here, all minor objects restricted to the molecule, such as atoms or bonds, are also deleted, as well as all minor objects which refer to deleted atoms, for example bonds, rings and groups which partially or fully overlap the atoms of the deleted molecule.

Object duplication will trigger a *dup* property invalidation event in the new ensemble. Properties such as unique IDs may be defined to not survive duplication even if there are no structural changes in the objects they are linked to. Deleting a molecule, or merging ensembles, will trigger a *merge* invalidation event. Again, properties may be set up to react to this event.

There are no similar functions for rings or ring systems. These are intended to be managed completely by the system. Groups do have deletion commands - but in that case, the deletion operation applies only to the group object. No atoms contained in the group are deleted if the group is removed.

Groups

Groups are a generic mechanism to manage groups of atoms and/or bonds. In contrast to other minor object of ensembles they are intended to be managed by the application. There are fewer automatisms for maintaining groups and group data than, for example, molecules and rings.

The basic objects which forms a group are atoms and possibly also bonds. However, in addition to atoms, groups may contain other groups. This nesting can be of arbitrary depth, but no cyclic graphs must be produced.

Automatic group set-up

There is an automatic set-up mechanism for groups, but this function will simply initialize an empty group set.

Example:

```
ens groups [ens create CC]
```

will report an empty list.

Creation and modification of simple groups

Groups can be created, changed and deleted with a standard set of commands:

```
set glabel [group create $ehandle {1 2}]
group add $ehandle $glabel 3
puts [group get $ehandle $glabel G_SIZE]
group remove $ehandle $glabel 1
group delete $ehandle $glabel
```

This short sequence of commands show the most important methods to set up, modify and delete groups. The first line creates a group which contains the atoms of the ensemble with the labels 1 and 3. The commands returns the group label of the new group. The next line then adds atom 3 to the group. This kind of operation does not change the group label. The standard property data request for group property `G_SIZE` reports 3. Individual atoms may be removed from a group, and groups may be deleted, as shown in the final two lines of sample code.

Atoms may be a member of any number of groups, and they may be listed more than once within a group.

Groups and substructure matching

In an alternative way of defining groups, they may also be generated as a side effect of a successful substructure match command.

Example:

```
set st [ens create {C[N+](=O)[O-]}]
set ss [ens create {N(=O)=O} smarts]
match ss -creategroup 1 $ss $st
```

This example will set up a group with the atoms of the matched nitro group on the structure. The substructure match routine is smart enough to recognize the equivalence of the two nitro group forms. Using appropriate match options, it is possible to mark all instances of a substructure by generating a group for every match.

Every successful match will add more groups, so in case a structure is used for multiple matches and the accumulation of groups is not desired, they should be removed all prior to matching:

```
group delete $st all
```

Recursive groups

Groups may contain other groups, In case a group is deleted which is contained in another group, the containing group is also deleted. Recursive groups may be set up and modified by script commands, just as normal groups:

```
set glabel1 [group create $ehandle {1 2}]
set glabel2 [group create $ehandle [list [list "group" $glabel1] 3 4]
puts [group atoms $ehandle $glabel2]
puts [group objects $ehandle $glabel2]
group delete $ehandle $glabel1
puts [ens groups $ehandle]
```

This sample code first creates a basic group with atoms 1 and 2. Then, a second group is created which contains the basic group, and atoms 3 and 4. The distinction between atoms and groups as member objects is made by prefixing the group label by the object class identifier *group*. If no object class identifier is used, the object label is assumed to describe an atom. It is also possible to explicitly prefix atom labels by an *atom* object class identifier. The newly created recursive group is assigned a label, just as a standard group. The atom list of the recursive group will only list atoms 3 and 4. For a full member object listing, the objects command is available. In this case, it will report

```
{group 1} {atom 3} {atom 4}
```

The format is a fully qualified object list, which could be used in a **`group create`** statement. Deleting the simple group which is included in the recursive group will also destroy the recursive group, as demonstrated by the final statement which returns an empty list.

In case all atoms which are a member in a recursive group are needed, directly or indirectly as part of an included group, a simple recursive function can be used:

```
proc all_groupatoms {ehandle glabel} {
   set alist [group atoms $ehandle $glabel]
   foreach glabel [group groups $ehandle $glabel] {
      set alist [concat $alist [all_groupatoms $ehandle $glabel]]
   }
   return $alist
}
```

Recursive groups and 3D searching

Recursive groups were originally introduced for the implementation of 3D structure searching. In 3D structure searching, coordinates of structure features which need to be matched are often defined in a dependent fashion. For example, the centroid of a matched ring should be within a certain distance to another atom.

In the toolkit, this is modelled by using groups and recursive groups as part of the substructure to represent these relationships. For example, the distance constraint is encoded as a substructure group containing an atom and the centroid group, which contains the atoms of the ring. When a match is checked, coordinates of matched *structure* fragments are accessed by the *substructure* groups. The distance can only be computed after the centroid coordinates have been established, and these can only be obtained if the ring substructure atoms where matched onto structure atoms with defined coordinates.

These group hierarchies are set up automatically when, for example, an ISIS 3D query file is read. But since the mechanism is general, it is also possible to configure it manually.

Example:

```
set ss [ens create c1ccccc1.N smarts]
set centgroup [group create $ss {1 2 3 4 5 6}]
set distgroup [group create $ss [list [list “group” $centgroup] 7]
group set $ss $centgroup G_CONSTRAINT centroid
group set $ss $distgroup G_CONSTRAINT [list distance [list 3.0 4.5]]
```

This substructure can now be used for 3D matching and will only match those 3D structures where the distance between the centroid of the phenyl ring and the nitrogen atom is between 3 and 4 Angstroms. The extraction of coordinate information and the checking of the 3D constraints is automatically handled within the substructure match routine.

Traps and Pitfalls

There are two cross-referencing commands which are very similar, but perform clearly distinct operations:

```
group group $ehandle $gspec
```

will return the group label from an alternative group specification, such as an index or a property value look-up. This is the same mechanism as in the **`atom atom`** or **`bond bond`** commands.

```
group groups $ehandle $gspec
```

The only typographical difference is the plural s in *groups* (vs. *group*). This command lists all groups with a a member of the group, which is always a different result since a group cannot contain itself.

String Representations of Structure Data

The **CACTVS** toolkit supports several methods of encoding and decoding structure and reaction information as strings.

SMILES

CACTVS supports **SMILES** nearly completely. The only unsupported feature is the encoding of higher-order stereochemistry (square planer, pentagonal bipyramid, octaeder). After the SMILES structure code part, separated by whitespace, an optional name may be added. This information is stored in property `E_NAME` after decoding. **SMILES** strings may be stored and generated as property `E_SMILES`.

The toolkit contains an implementation of the original version of the **Unique SMILES** algorithm. However, the published version is no longer identical to what Daylight is actually using in its software. It is possible to generate **Unique SMILES** from an ensemble with this toolkit, but it does not match the results of Daylight software for almost any non-trivial structure. For structure comparison, we strongly recommend the use of the native **CACTVS** structure hash codes instead of **Unique SMILES** strings.

The **E_SMILES** property has the following computation parameters which influence the style of the result string:

• *usearo* (default 0)	If set, aromatic atoms will be output in lower case, and double bonds in aromatic systems will not be output explicitly as a Kekulé system. The default output style is as a Kekulé system with fully specified bond orders and element symbols starting with an uppercase letter.
• *useisotope* (default 1)	If set, atomic isotope information (property `A_ISOTOPE`) will be encoded in the **SMILES** string if it is available. If this flag is not set, isotope labelling information will be ignored even if it is present.
• *usemapping* (default 1)	If set, atom mapping information (property `A_MAPPING`) will be encoded in the **SMILES** string if it is available. If this flag is not set, atom mapping information will be ignored even if it is present.
• *usesmarts* (default 0)	Encode as SMARTS, with explicit hydrogen counts at all atoms, and #1 notation for hydrogen atoms which are not encoded as hydrogen counts of core atoms.
• *usestereo* (default 1)	If set, atom and bond stereochemistry (properties `A_LABEL_STEREO` and `B_LABEL_STEREO`) will be encoded in the **SMILES** string. An attempt will be made to compute `B_LABEL_STEREO` if it is not present, but not for `A_LABEL_STEREO`. If the computation fails, no error is generated and stereochemistry is not output. If this flag is not set, stereochemistry will be ignored if it is present.

• *usesuperatom* (default 0)	If set, super atoms will be included in the **SMILES** string with their symbol. This can result in illegal **SMILES** strings. If the flag is not set, super atoms in the ensemble are ignored.
• *unique* (default 0)	If set, the result string will be Unique **SMILES** according to the original publication.

Example:

```
set ehandle [ens create c1ccccc1]; ens new $ehandle E_SMILES {} {usearo 0}
```

This sample code will first decode an ensemble from a **SMILES** string. The original **SMILES** string is stored as property `E_SMILES`. The next statement re-computes the **SMILES** string, but with the computation parameter *usearo* set to 0. The result is "*C1=CC=CC=C1*".

SMARTS

Cactvs supports nearly the full **SMARTS** feature set, including **Recursive SMARTS**. The only exception is again high-order stereochemistry.

Reaction SMILES

Reaction SMILES is fully supported. The optional middle part of a reaction specification will be decoded as a reaction ensemble with property `E_REACTION_ROLE` set to *agent*. Reaction **SMILES** strings may be stored and generated as property `X_SMILES`. Optional atom mapping labels in Reaction **SMILES** expressions cannot be negative numbers. Atom mapping labels will be deposited and read from property `A_MAPPING`.

Example:

```
set xhandle [reaction create {[CH2:1]=[CH2:2]>[Pt]>[CH3:1][CH3:2]}]
set xmiles [reaction new $xhandle X_SMILES]
```

The first sample line creates a reaction with three ensembles. By looking at property `E_REACTION_ROLE` they can be identified via their roles of *reagent*, *product*, and *agent*. The ensembles are really always listed in the reaction in that order when they are added by the **Reaction SMILES** decoder, but this is not a sequence which should be relied upon for general reaction processing. The second line re-computes the **Reaction SMILES** string, which in this case is completely identical to the input string.

SMIRKS

The toolkit supports the use of SMIRKS for the description of reaction transforms. The advanced version is supported - i.e. the creation and deletion of atoms within a transform is supported.

However, stereochemistry change, both on atoms and bonds, is not yet possible. The only atom attribute change which is already fully supported are changes in formal charge. The creation, deletion and change of bonds within a transform is completely implemented.

Example:

```
set ehandle [ens create C=C]
set thandle [reaction create {[C:1]=[C:2]>>[C:1]-[C:2]} smirks]
ens transform $ehandle $thandle
```

SMILES and SMARTS extensions

A number of useful backward-compatible extensions were introduced into the **`SMILES`** and **`SMARTS`** decoders:

Attribute ranges

At all places where an attribute count is expected, a range may be specified instead. Ranges are enclosed by curly braces. They may be open on either or both sides, with an implicit lower limit of 0 and upper limit of 31.

Example:

```
ens create {[C;H2,H3,H4]} smarts

ens create {[C;H{2-4}]} smarts

ens create {[C;H{2-}]} smarts
```

The first example is standard **`SMARTS`**. The other lines show how to use ranges for more compact and readable encoding.

Implicit superatoms

If an atom cannot be decoded as an element symbol, or, in a SMARTS context, as a SMARTS expression, a superatom will be created. This feature is an option of the decoder and not active in all contexts. The superatom symbol will be stored in property `A_SUPERATOMSTRING`.

Example:

```
ens create {[Boc]}
```

Explicit superatoms

Atom symbols starting with a tilde character ~ in a bracketed atom expression are decoded as superatoms. The superatom symbol (`A_SUPERATOMSTRING`) does not include the tilde. The superatom symbol may consist of digits and letters, plus the underscore and minus characters. The superatom may possess additional attributes or be a port of an SMARTS expression, but these must be separated by explicit logical expression operators, such as ',' or ';' because the usual rules of tokenization are not used in order to determine the end of the superatom symbol.

Example:

```
ens create {[~COOH]}
```

Special atoms

The following special atom types are introduced:

`HA`	Generic hydrogen acceptor. This checks property `A_HYDROGEN_BONDING`.
`HD`	Generic hydrogen donor. This checks property `A_HYDROGEN_BONDING`.
`D`	Deuterium. This checks `A_TYPE`, `A_ELEMENT` and `A_ISOTOPE`.
`T`	Tritium. This checks `A_TYPE`, `A_ELEMENT` and `A_ISOTOPE`.

Attributes

The following extra query attributes and attribute extensions are recognized:

a

If used without a count, it corresponds to the standard **SMILES** meaning of *aromatic*. In this toolkit, this attribute can optionally take a count which is interpreted as the number of aromatic rings the atom is a member of.

Example:

```
ens create {[C;a{2-}]} smarts
```

This defines a carbon atom which is a member in two or more aromatic rings, for example the two centre atoms of naphthalene. The checked property is `A_ARORING_COUNT` in the extended case, `A_ISAROMATIC` in the standard case.

e

Number of π-electrons in a ring the atom is member of. The checked property is **R_N_PI_ELECTRONS**.

Example:

```
ens create {[S;e6]} smarts
```

X

If used with a count, it is the standard **SMILES** neighbour count. If used without a count, it defines a hetero (no carbon, no hydrogen) atom.

Examples:

```
   ens create {[X]} smarts
   ens create {[X2]} smarts
```

The first example will match any hetero atom. The second example matches any atom which has exactly two neighbors. In the standard case, the checked property is **A_NEIGHBORS**. In the extension case, an expression involving **A_TYPE** and **A_ELEMENT** is generated.

x

Number of hetero-atom substituents on an atom.

Example:

```
ens create {[C;x2]} smarts
```

This will match a carbon atom which has two hetero neighbors. The checked property is **A_HETERO_SUBSTITUENT_COUNT**.

Special bond types

A bond described solely by an exclamation mark is interpreted as a bond which must not be present in a substructure match (the decoded `B_TYPE` property value is set to *nobond*).

Example:

```
ens create {C!C} smarts
```

is a (disconnected) substructure where the two substructure atoms must not be matched on adjacent structure atoms, but

```
ens create {C!-C} smarts
```

is a substructure where the carbon atoms are linked by a bond which is not a single bond.

Non-overlapping recursive SMARTS fragments

By default, the substructure which is part of a Recursive SMARTS definition has no knowledge which part of the structure was already matched by the basic part of the substructure, and it is free to match on any atom in the substructure, even if it is already covered by another substructure atom in the hierarchy above.

This toolkit allows the exclusion of the part of the substructure which was already matched. This feature can be activated by using a double $$ as initiator of a recursive SMARTS specification instead of a single $.

Example:

```
set ss1 [ens create {CN[$(CN),$(CO)]} smarts]
set ss2 [ens create {CN[$$(CN),$$(CO)]} smarts]
match ss $ss1 CNC
match ss $ss2 CNC
```

The first SMARTS structure is a classical Recursive SMARTS definition. The second one uses the same fragments but the extended syntax to prevent the recursive fragments to match on any structure part which was already matched. On the simple sample compound dimethylamine, the first match succeeds with the C-N fragment, because the carbon fragment atom matches on the second carbon atom, and the nitrogen fragment atom matches onto the central nitrogen atom, without any knowledge that this atom was already matched by the nitrogen atom in the base fragment. The same match with the extended syntax fails, because the nitrogen fragment atom cannot be matched - the central nitrogen atom in the structure is not allowed, because it is already covered by the base fragment.

Exclusion atoms

Negated recursive fragments are often not very useful, because they tend to be matched on unexpected structure parts.

Example:

```
set ss [ens create {C[!$(Cl)]} smarts]
match ss $ss CCl amap
```

This match attempt will actually succeed, because the substructure atom which should be *not chlorine* is matched onto one of the hydrogen atoms in the structure. The atom match map in variable **amap** will contain the list "{1 1} {2 3}", showing that the first atom of the substructure was matched on the first atom of the structure (carbon on carbon), and atom 2 (the *not chlorine* atom) on atom 3 (one of the hydrogen atoms).

For atoms and fragments which should not match in any way, the CACTVS toolkit implements the concept of exclusion atoms and fragments. Exclusion atoms are specified by the attribute ^. Exclusion fragments are substructure fragments which consist only of exclusion atoms.

Example:

```
set ss [ens create {C[^Cl]} smarts]
match ss $ss CCl
```

This match attempt will *not* succeed. Matches with exclusion atoms will fail whenever these atoms can be matched in some way. For this purpose, all exclusion atoms which are linked together by bonds are treated as a joint fragment.

Example:

```
set ss [ens create {C[^O][^C]} smarts]
match ss $ss COC
match ss $ss CO
```

Here, the first match attempt will fail, because the carbon atom contains an OC substituent, which is prohibited by the sequence of exclusion atoms. The second attempt will however succeed, because there is no way to match the whole group of exclusion atoms.

Groups of exclusion atoms are processed after all non-exclusion atoms have been processed, and at that moment they are treated as substructure extensions where each branch must be assigned a location - only that a failure to find such a location is considered a success.

Example:

```
set ss [ens create {[^C]O[^C]} smarts]
match ss $ss COC
match ss $ss COO
match ss $ss CO
```

The first match attempt will fail, because all independent exclusion groups could be assigned matches. The second and third examples will match, because it is not possible to assign both non-carbon exclusion branches independent locations since there is only one carbon, and overlaps of exclusion fragments are not allowed.

In case of ambiguities in the string syntax, the standard interpretation has precedence. So, [Se] is a standard selenium atom, and [S;e] a sulphur atom which is a member of a ring with one π electron. The only exception is the pseudo atom HA - conceivably, that atom could be interpreted as an aliphatic atom with one hydrogen neighbour.

SMILES and SMARTS traps and pitfalls

SMILES and SMARTS are easy to use and powerful and play a prominent role in typical script applications developed with the CACTVS toolkit. However, there are some problems which appear to be encountered frequently. The following section describes some common errors and correct solutions which address these problems.

SMILES/SMARTS decoder mode

The decoder mode influences the interpretation of the structure string. Example:

```
ens create c1ccccc1 hadd

ens create c1ccccc1 nohadd

ens create c1ccccc1 smarts
```

These three decoder statements produce three clearly different internal structure representations. The first (which is the default which is also used if no decoder mode is explicitly set) produces a benzene molecule complete with hydrogen, with alternating single and double bonds in a Kekulé structure. When used as a substructure, this structure will *not* match arbitrary phenyl rings, except benzene, because the hydrogen atoms are part of the structure and need to be matched too.

The second line is similar, but no hydrogens are added, so only six carbon atoms are generated. This structure will match phenyl rings - but in the default match mode, where aromatic bonds match both single and double bonds, will also match a hexane or hexene ring.

Only the last sample line is a complete SMARTS decoding - here, the match attribute *aromatic* is set as an atom flag on all the carbon atoms, because the atom symbols are specified in lower case. Since these carbon atoms, if used as a substructure, will only match aromatic structure atoms, this substructure specification will *not* match hexane or hexene.

Hydrogens

CACTVS generally wants to work with structures where all hydrogen atoms are defined. Property computations will often report misleading results if these routines are called with hydrogen-depleted structures. Therefore, hydrogen atoms in SMILES are expanded when a structure is created from the string. This is not the case in SMARTS decoding, and additionally, the H attribute in SMARTS is a *hydrogen count,* while in SMILES it specified a hydrogen atom. The alternative notation via the element number must be used in case explicit hydrogen atoms should be generated from SMARTS string. Note that explicit hydrogens such as in [NH2] or [#1] are always expanded, even if the *nohadd* decoder mode is used. In order to get rid of these atoms, an *ens hstrip* command can be executed.

```
(1) ens create {[NH2]}

(2) ens create {[H][N][H]}

(3) ens create {[N;H2]} smarts

(4) ens create {[NH2]} smarts

(5) ens create {[H][N][H]} smarts

(6) ens create [[#1][N][#1]} smarts
```

The first line decodes into a nitrogen atom with two hydrogen atoms attached. Since the number of hydrogens is explicitly specified on all atoms, the selection of the *hadd* (default) or *nohadd* decoder modes (but not *smarts*, see examples below) does not make any difference. In both modes, the second line decodes to exactly the same structure, the only difference being the order of the atoms in the atom list.

Line three is where things begin to become interesting. This line will decode into an ensemble with a nitrogen atom with the additional query constraint of needing to have exactly two hydrogen neighbors. If the structures defined in line one or two are used for substructure matching, they will match ammonia (NH_3), mapping the two substructure hydrogen atoms onto two of the structure hydrogen atoms and leaving the last structure hydrogen atom unmatched. The substructure generated in line three will *not* match, because the nitrogen atom in that structure must possess exactly two hydrogen neighbors.

Line four demonstrates a CACTVS SMARTS decoder extension. In original SMARTS, this encoding is completely equivalent to that of the previous line, but CACTVS introduces a subtle difference. If an element symbol is directly followed by a hydrogen count, without any logical operators, or other query attributes between these atom definition components, the hydrogen atoms are instantiated. This substructure string will expand into three atoms - one nitrogen and two hydrogens. The nitrogen atom will still bear the constraint that in substructure matching it must possess exactly two hydrogen neighbors. However, this substructure is able to provide explicit substructure/structure atom correspondence information for the hydrogen atoms, which can be useful.

SMARTS encodings as shown in line five are usually written unintentionally. This string defines a substructure of three atoms - a nitrogen atom and two *any* atoms, which both must possess exactly one hydrogen neighbour. H is the *hydrogen count* attribute in SMARTS, not a hydrogen atom. If no element is specified as part of the atom definition, it is translated into an *any* match atom which then used to attach additional constraints. Syntactically this encoding is absolutely correct, and the query will work as defined in the SMARTS standard, but the results may be unexpected, since [H] does *not* encode a hydrogen atom. Line six implements correctly what may originally have been intended by the construct in line five: Two explicit hydrogen atoms, and a nitrogen atom as a simple SMARTS substructure without additional attributes, as in line one or two.

Serialized major object strings

Major objects, such as ensembles, reactions, networks, tables and datasets, may be packed into a string. This string captures the full state of the object (including, by default, all subobjects). If no filtering operations are used to restrict the type of data which is packed into the string, it is a lossless method of encoding the object state. Technically, this string is a base-64 encoded, zlib-compressed, serialized object, using XDR encoding for platform-neutral storage of byte-order dependent data. After removing the encoding and compression layers, these strings are very similar, but not identical, to the native CACTVS binary file formats for I/O of these objects.

Since the base64-encoded packed string is guaranteed not to contain any non-printable or problematic (quotation, etc.) characters, these strings can be conveniently sent by mail, stored in database columns, etc. Packed object strings are portable and platform-independent.

Only major objects may be packed into packed formats.

Example:

```
set packstring [ens pack [ens create CCC]]
set newens [ens unpack $packstring]
```

Packed object strings are significantly larger than, for example, a molecule representation as a SMILES string. However, this string will preserve all attached ensemble and minor object data, such as 2D and 3D coordinates and other properties, which cannot be done with SMILES, SLN or other simple string encodings.

SMILES and SMARTS files

Both SMILES and SMARTS, including the reaction versions, can be saved to and read from structure files. This is a built-in file format. The SMILES/SMARTS format is automatically detected, including whether it contains reactions or structures. It is not possible to read Reaction SMILES files in ensemble or molecule input mode, and vice versa. During the format analysis procedure, the read mode is set appropriately and should not be changed.

In the simplest case, SMILES/SMARTS files are simply text files which contain a SMILES/SMARTS string on each line. After the actual structure code, a name field (property `E_NAME`) may follow separated by whitespace.

Files are by default read with the addition of implicit hydrogens, just as it is the default when decoding a SMILES string with an **`ens create`** statement. This is independent of the explicit hydrogen addition or removal set as a file attribute. The file-wide hydrogen processing is applied to every ensemble read from the file handle regardless of the file format. The implicit hydrogen addition when reading SMILES files is the result of a flag set on the decoder which is invoked as part of the record input process, long before the postprocessing operations. In case this addition of implicit hydrogen is not desired, for example when reading substructures from file, it can be disabled by the statement

```
molfile append $fhandle flags noimplicith
```

Besides structure data lines, SMILES files may contain the following types of additional lines without raising an error:

Empty lines

These are silently ignored

Comment lines

These are lines starting with an '#' character. They are ignored.

File property lines

These are lines beginning with the magic character sequence '#F' and containing a property name and a property value. When a line like

```
#F content New lead series
```

is read, a property `F_*CONTENT*` is attached to the file object (or `F_CONTENT`, in case that property is defined) and set its value to the trimmed remainder of the line ("*New lead series*" in this case). This data can be retrieved with a

```
molfile get $fhandle F_*CONTENT*
```

command. When the file is opened, all file properties which occur before the first structure line or ensemble property line are immediately available without reading any records. Later file property data becomes available as soon as the line has been read as a result of normal structure input.

Ensemble property lines

Similar to file property lines, these are lines which begin with the magic characters '#E'. All ensemble properties which are found *before* the actual structure line is read are attached to the ensemble as regular property data, analogous to the processing of file properties. If a set of lines like

```
#E E_IDENT PharmaconX123
CCO
#E E_IDENT PharmaconX124
CCS
```

is encountered, the property `E_IDENT` (here specified in **CACTVS** nomenclature, and since this is a defined property, no name standardization takes place) is attached to both records. The first read command will retrieve ethanol (CCO) with `E_IDENT` set to *PhramaconX123*, and the second input operation fetches CCS and sets its `E_IDENT` to *PharmaconX124*.

Folded lines

In an attempt to counter a common problem in e-mailed **SMILES** files, the input routine will attempt to detect the folding of long lines in the original file into two shorter lines by a mail tool or another processing application. If a structure could not be successfully decoded from a single line, an attempt is made to join the next line to the first line and decode that the joined line. This is not foolproof, but often works as a reasonable makeshift auto-correction mechanism. Only a single line will be merged. If the file is even more broken, its problems should be fixed at the source.

Indented lines

The indentation level (number of whitespace characters) of **SMILES** and **SMARTS** data lines is read into property E_LEVEL. Thus, a file structure like

```
CC[Cl,Br,I]generic
CCClchlorinated
CCBrbrominated
CCIiodated
```

can be conveniently used to store hierarchies. The first line will be read with `E_LEVEL` set to 0, and the other lines (assuming a tab character is used) with `E_LEVEL` set to 1. If `E_LEVEL` is set for an ensemble when it is written to a **SMILES** file, indentation will automatically be added. If this property is not present, no indentation is used.

Structure names

Everything to the right of the first word of the data lines (which is decoded as the **SMILES** string proper) is read as structure name and stored in property `E_NAME`. Example:

```
CCC propane
```

If this line is read, the structure is decoded, and its name (property `E_NAME`) is set to *propane*. The name part will be trimmed on both sides, but not split. Any whitespace after the beginning of the name part and not reachable by an uninterrupted sequence of whitespace characters from the end of the line will be preserved. Example:

```
CCC propane 74-98-6
```

Here, the name will be set to "*propane 74-98-6*". Further processing of the name will have to be performed by explicit script commands. Note that indexing of string words can be very helpful for this task, as in

```
ens set $ehandle E_CAS [ens get $ehandle E_NAME(1)]
```

Other structure string representations

The toolkit supports the work with a number of additional string representations:

Hex-encoded SMILES

In standard structure decoding contexts (such as in an ens create command, but not while reading files), an attempt is made to interpret a string not just as a SMILES string or a serialized object string, but also as a hex-encoded SMILES string if the first two methods fail. Hex-encoded SMILES is used in a number of Daylight tools.

Example:

```
ens create [encode -hex CCC]
```

PubChem Compound IDs (CIDs)

A simple integer is interpreted to represent a PubChem compound ID. The structure is looked up on the interpet.

Example:

```
ens create 1
```

Cactvs Minimols

A **Cactvs Minimol** is an extremely compact representation of a structure with all attributes which are usually of relevance for structure searching. In contrast to serialized object strings, this format does not encode arbitrary data, but only a fixed set. Compared to SMILES and similar formats, the information density is much higher, and decoding much faster because for standard structure matching o proeprties need to be computed. Minimols can be computed as property `E_MINIMOL`, and decoded direclty in **`ens create`** statements.

Example:

```
set mm [ens get [ens create CCC] E_MINIMOL]
set eh [ens create $mm]
```

CAS Numbers

Cactvs recognizes CAS numbers and will attempt to decode them by means of a PubChem lookup.

Example:

```
set cas [ens get [ens create CC(=O)C] E_CAS]
set eh [ens create 67-64-1]
```

This is a rather expensive operation and slow for larger sets of compounds.

Sybyl Line Notation (SLN)

SLN is fully supported, but not as a built-in format. Encoding and decoding of SLN strings must be performed via string file operations.

Example:

```
filex load sln

set sln [molfile string [ens create CCC] format sln]

set fhandle [molfile open $sln s]

set ehandle [molfile read $fhandle]

molfile close $fhandle
```

In this code sequence, first the SLN I/O handler is loaded (in most toolkit versions, it is not a built-in format). The second line shows how to generate an SLN string by using the **`molfile string`** command. The last three line demonstrate the decoding of an SLN string by first opening the string as a file (mode *s*), reading the first record from the string file, and finally closing the structure file handle.

JME strings

JME, the native format of the popular JME Java editor applet by P. Ertl of Novartis, is also fully supported as an external I/O module.

Work with this format follows the same procedures as the SLN case.

Wiswesser Line Notation

There is a standard property definition for WLN data called `E_WLN`, but currently neither encoding nor decoding of this format are supported.

Envelope encodings of SMILES and other structure strings

The toolkit contains utility commands for encoding and decoding arbitrary strings into and from various encoding formats. The **`encode`** and **`decode`** commands are fully explained in the auxiliary command section.

Examples:

```
molfile read [set fh [molfile open [decode -zip64 $z] s]]; molfile close $fh

ens create [decode -url $z]

puts "<A HREF="[encode -url /cgi/bin/myapp.cgi?smiles=$z]\">"
```

The first line shows how to read a *zlib*- or *gzip*-compressed, base64-encoded string which is an image of an arbitrary structure record, for example an MDL Molfile. The second lines demonstrates the decoding of an URL-encoded SMILES string. The final example shows how strings with characters which need to be protected, such as ’#’ or ’&’, can be output in the context of a CGI Web application.

Property Data Types

All property data has a defined data type. The data type determines how the information is internally managed. All memory, file handles, etc. which are required by a data type are handled by the system and are automatically released when property data is deleted, regardless whether it is a direct discard, or an indirect effect by the deletion of objects holding the data or the result of an invalidation because of object relationship changes.

Data type handlers

Every data type is associated with a handler module. This handler module provides a predefined set of functions, such as decoding, duplication, deletion, and output formatting. Handlers for the most common data types are built-in. Additional handler modules may be loaded at runtime. This may happen either explicitly, or indirectly by referring to a property which declares itself in its description record to be of a data type which the system does not yet know about. In that case, a handler module is automatically looked up. If it cannot be found, the loading of the property definition fails.

Examples:

```
typex load floatvolume
```

```
filex load gausscube; set ehandle [molfile read test.cub]
```

The first command explicitly loads the handler for floating-point volume data, which is not in the built-in set. The second line does it indirectly: First the I/O handler for *Gaussian* cube files is loaded and then a sample cube file is read. The input routine tries to attach the volume data found in the file to the ensemble receives the input data, using the float volume data type. At the moment the input routine refers to property `E_VOLUME`, its definition is looked up, and when the property definition file for `E_VOLUME` is read, a handler for its data type `P_FVOLUME` is located and loaded. Both the paths for property look-up and for I/O handler look-up can be configured on the scripting language level via the global **`cactvs()`** control array variable.

Storage slots

Property attached to objects is kept in slots of a fixed size. If the data is of variable size, part of the slot structure is used to contain a pointer to allocated memory. Because it is more efficient to use only the slot structure, there are several data types for common storage requirements such as float pairs (used for example for 2D display coordinates) which could be stored as a vector, but are implemented as a separate data type. 3D coordinates, even though they always require just 3 dimensions, are manage by the standard float vector type - since three floating point values do not fit into the slot structure, there is no notable efficiency gain in providing a specialized 3-element vector type. Since the slots contain room for both a data pointer and a length field, vectors may be of variable and non-uniform length on each individual data item.

Internal and external representation

In some cases, there is a distinction between the internal and external representation of property data. For example, internally there is only a single simple floating point type, which stores the data as a *double* value. However, export functions may distinguish between a single-precision float and a double-precision float, and consequently these two data types have separate output functions.

Similarly, various integer types are internally all stored in a long value, but may be output as *boolean*, *byte*, *short* or *long* values.

Naming conventions

In the scripting environment, data types are usually addressed by their readable name. data types also have a system name, which can be used as an alias, but usually it is more convenient and more readable to refer to the floating point volume data type as *fvolume* instead of `P_FVOLUME`. The latter is the system name which is for example used in property description files.

Subscript names for data with identifiable fields are always spelled in lower case. They must not contain whitespace or punctuation characters.

Built-in data types

The data types in the standard built-in set are:

• boolean	Internally a long integer, this type as import and export functions for encodings such as T/F, and may be stored as a bit or byte on compact formats. This type is not indexible.
• byte	An 8-bit signed integer. Internally managed as a long integer and not indexible.
• short	A 16-bit signed integer. Internally managed as a long integer and not indexible.
• int	A 32-bit signed integer. Internally managed as a long integer and not indexible.
• uint8	A 64-bit (8-byte) unsigned integer, which is primarily used for hashcodes. Not indexible. The standard method of input and output for this data type is as a 16-character hex string, not a decimal number.
• float	A single-precision (32-bit) floating point number. Not indexible. Internally stored as a double precision float.
• double	A double-precision (64-bit) floating point number. Not indexible.
• string	An arbitrary-length zero-byte terminated ISO-string. It may contain any character, including control characters such as linefeeds, except the zero byte. This type can be indexed on a word basis.
• unicode	An arbitrary-length Unicode string. Internally, it is encoded as wchar_t platform-dependent characters, with a zero byte word as terminator.
• shortstring	An optimized string with a maximum length of 8 bytes (16 on 64-byte platforms, but I/O will be limited to 8 characters). Numeric field indexing refers to the character position, not the word as in normal strings.

• index	This type contains 4 unsigned short integers with a special function. The four integers are used to encode group memberships and membership numbers on the ensemble and molecule levels. There are two number pairs, one for the ensemble and one for the molecule attribute. The first number of each pair is the class number, the second the instance number within that class. The fields may be directly addressed by the predefined subscript names *eclass*, *ecount*, *mclass* and *mcount*.
• bitset	This is a bit set with a maximum of 32 positions. The names of the positions are specified in the property enumeration field. Using these names, bits may be addressed individually in a property-dependent fashion. Alternatively, a numeric subscript in the range 0...31 can be used.
• intpair	A pair of two 32-bit signed integers. They may be individually addressed with the predefined subscript names *x* and *y.*
• intquad	A quartet of four 16-bit signed integers. They may be individually addressed with the predefined subscript names *x1*,*y1*,*x2* and *y2*.
• floatpair	A pair of two single-precision floating point numbers. They may be individually addressed with the predefined subscript names *x* and *y.*
• qualifiedint	This is an integer with additional precision, validity and range information. Internally, it consists of a base value, a qualifier (by default, *eq*, other possible qualifiers are *le*, *lt*, *ge*, *gt*, *approx* and *missing*), and positive and negative integer deviation ranges. If the qualifier is *missing* or *null*, the default output is *N/A,* and the value or deviations are ignored. If the qualifier is *eq*, and no ranges are set, the display format is the same as for an integer. If the qualifier is not *eq* and not *missing*, the output is the base value, followed by the identifier, and then the value range. If positive and negative deviations have the same value, the range is displayed as *+/-range*, otherwise as a pair of negative and positive deviations with explicit signs. For retrieval, specific subfields *value*, *qualifier*, *+delta, -delta, delta*, *lowbound* and *highbound* can be used to obtain specific information. *delta* is the sum of positive and negative deviations, *lowbound* the base value minus the negative deviation, and *highbound* the base value plus the positive deviation. For input, a qualified integer may be specified as a simple integer, a full specification of all fields, or several abbreviated forms.
• qualifiedfloat	This datatype is very similar to the qualified integer type, except that the base value and the deviation ranges are floating point numbers.
• blob	A raw data block of variable length. The data may contain any byte values, including zero bytes. It is indexible via a numeric subscript allowing access to individual data bytes, which are returned or set as unsigned bytes.

• date	This type describes a date, with a precision of one second. The value used for normal retrieval is an ISO time value (YYYY-MM-DD HH:MM:SS). For setting, the conversion routines understand a couple of standard date formats and will detect them automatically. Additionally, the raw numerical system time value (seconds since 1970), and reserved words such as *now* and *tomorrow* are understood. When a toolkit version is providing Tcl scripting capabilities, it will use the rather capable Tcl time decoder in addition to scanning its private lists of standard formats. In case the raw numerical value of a date item is needed for custom formatting, for example with the Tcl **`clock format`** command, the *nget* command should be used. For retrieval only, the reserved subscripts *year* (including century), *month* (0-11), *day* (1-31), *hour* (0-23), *minutes* (0-59), *seconds* (0-59), *weekday* (0-6, week begins on Sunday) and *yearday* (0-365) give access to specific time unit values.
• url	This data type encodes an URL. In most respects, it is similar to the *string* data type, but instead of word indexing, it may be indexed for read access only by the reserved subscript names *directory* (directory part of pathname, assumes fully specified path name or terminal /), *file* (filename part of pathname, assumes fully specified path name or terminal /), *hash* (page location), *host* (hostname:port combination), *hostname* (pure hostname without port), href (full url), *ip* (host IP address), *password* (password portion of the access credentials, if path of the url), *pathname* (path to the object on the server, excluding *host*, *password*, *user*, *hash*, *search* fields), *port*, *protocol*, *search* (query part of url), *target* (always empty), *text* (always empty), *user* (user portion of access credentials, if specified in the URL), and *ipaddr* (resolved host name). These field names are the same as in the JavaScript *link* object, plus some custom additions (*user*, *password*, *ipaddr*). In some output contexts a hyperlink will be written instead of the string data value, for example in HTML table output. This data type currently cannot store target and link text information. These fields will always be empty.
• choice	This is a wrapper data object which can hold items of different, but predefined datatypes. The field definition of the associated property definition declares possible values by associating a field name either with a primitive datatype (such as *int*), or a refer to a property (such as `E_NCBI_PUBLICATION_PATENT`). Using the latter construct, very complex nested datatypes of variable layout may be constructed, since the refererred properties may themselves be of a complex structure, including being of the *compound* and *choice* data types. The default output format is a list of the name of the field encoded in this object instance, and the value proper. In addition, subfields *value*, *datatype*, *property*, *index* and *name* may be used to retrieve specific aspects of a datum.
• bitvector	A bit vector of variable length. Individual bits can be selected via a numeric subscript. As in the *bit* type, bit positions may be also assigned names via the property enumeration values. and these names used for indexing

• bytevector	An unsigned byte vector of variable length. Individual elements can be addressed via a numeric subscript.
• shortvector	A signed 16-bit integer vector of variable length. Individual elements can be addressed via a numeric subscript.
• intvector	A signed 32-bit integer vector of variable length. Individual elements can be addressed via a numeric subscript.
• floatvector	A 32-bit single precision float vector of variable length. Individual elements can be addressed via a numeric subscript.
• doublevector	A 64-bit double precision float vector of variable length. Individual elements can be addressed via a numeric subscript.
• tensor	A double precision floating point tensor with 9 elements. Individual elements can be addressed via a numeric subscript. In most respects, this data type behaves like a normal floating point vector.
• xyvector	A 2D coordinate vector. Each element consists of a single-precision floating point pair. Numerical element indices will retrieve or set a pair, not individual numbers.
• stringvector	A variable length vector with string elements. The string elements are zero-byte terminated ISO-encoded strings of arbitrary length. It is possible to have **`NULL`** elements in the vector. Individual elements can be selected via a numeric subscript.
• unicodevector	A variable length vector with unicode elements. The string elements are *wchar_t*-encoded strings of arbitrary length with an all-zero-bytes stop word. It is possible to have **`NULL`** elements in the vector. Individual elements can be selected via a numeric subscript.
• uint8vector	A variable length vector with 64 bit unsigned integer elements. Individual elements can be addressed via a numeric subscript.
• compoundvector	A variable length vector with elements that are compound datatypes (see above in this paragraph).
• choicevector	A variable length vector with elements that are choice datatypes (see above in this paragraph).
• diskfile	A reference to a disk file. If the disk file refers to a file in a temp directory, the file will be automatically deleted when the property data elements gets deleted. This data type maintains an open file pointer to the file. Many platforms have limitations on the number of open file pointers, so this format should not be used when a large number of data instances are managed. If just the name of a file needs to be stored, it can be done as a string value. The default retrieval value of the *diskfile* data type is its path name. Additional information may be obtained by using the magic subscript names *name* (file name, same as default), *content* (data content), *size*, *format* (MIME type format, if known), *mode* (mode bits), *owner* (owner uid), *group* (group id), *readtime* (last file access,

st_atime status field), *writetime* (last file content change, **st_mtime** status field), *createtime* (last inode change, **st_ctime** status field). These indexed attributes are read-only. The time-stamp values are returned as ISO dates in a context where enumerated values are allowed, as seconds since 1970 otherwise.

- mapfile — Essentially the same as the *diskfile* type, but the content is kept in memory by memory mapping.
- dictionary — This is a hashed array of keyword/value pairs, similar to array variables in Tcl. Keys data values are arbitrary-length strings. The standard output is a list of keyword/value pairs (suitable for use with the Tcl **array set** command), and this is also the input format (which can be conveniently generated by the Tcl **array get** command). This type is indexible by the keywords which can be different in every data instance. For setting, specifying a non-existent keyword creates a new key/value pair, otherwise the old value is changed. For retrieval, a non-existent key results in an error.
- tree — This rather complex data type encodes a tree of nodes with the possibility to store a string value at each node. The standard input and output formats for the full tree are, beginning with the root node, a nested list of node name, node value, and the children as additional list elements, where every child node is recursively written as another list of the same format. Every node may be named, and individual nodes can be addressed with a name constructed from the names of the nodes beginning with the root node by concatenating them with a dot as separator character, such as in *root.node1.node2*. Alternatively, names using the children index number of each node may be used.
- query — This data type is closely related to the *tree* data type. It encodes query trees for complex atom and conditions. It is for example used in the query subfields of the **A_SEARCHINFO** and **B_SEARCHINFO** compound properties. Here, the inner nodes are logical operators (*and*, *or*, *not*), and the leaf nodes are property value comparison expressions. An example for a valid input or output string representation is "**and {A_ELEMENT = 6} {or {A_FORMAL_CHARGE <> 0} {A_SIGMA_CHARGE |>=| 0.3}}**", describing a match condition for a carbon atom which either bears a formal charge, or an absolute value of the Gasteiger charge of equal to or more than 0.3.
- compound — Compound properties are properties with multiple data fields, where each data field can have its own type. Typical examples are the standard properties for spectra which contain many fields in different formats. This type requires the fields to have property-dependent names. Individual fields may be accessed by using the field name, or a numerical index.

• structure	The data value is a molecular ensemble. Such ensembles are not part of the normal object set of the scripting environment and may not participate in datasets or reactions, but otherwise they behave like any other ensemble and show up in the list of registered ensembles. They possess property data of their own, have handles and may be manipulated via these. The value of this data type is the ensemble handle. This type is not indexible. Ensembles which are property data cannot be deleted by normal script commands. They disappear only when the property data they are part of is destroyed.
• reaction	Similar to the *structure* data type, this is a reaction as property data. Neither the reaction nor its ensembles may be deleted by normal means, nor are they part of the normal object set of the scripting environment. This type is not indexible.
• dataset	Similar to the *structure* or *reaction* data types, this is a complete dataset of ensembles and/or reactions which is property data and not part of the normal object set of the scripting environment. This type is indexible via the numerical dataset object list index in a read-only fashion.
• table	Similar to the chemical object data types, the value of field of this data type is a table object. It is not indexible, and the value for reading and setting the property is the table handle.
• network	Similar to the chemical object data types, the value of field of this data type is a network object. It is not indexible, and the value for reading and setting the property is the table handle.

The compilation environment has a mechanism to set up the set of built-in data types at compile time. For example, the *coorvec* extension module is often compiled into stand-alone applications.

Property-specific element subscript names

Additional subscript names may be provided by setting the fields property attribute. For example, atomic 3D coordinates may also be accessed by the indices x, y, and z, as in

```
atom get $ehandle $label A_XYZ(x)
```

which is possible only because the equivalent of the command

```
prop set A_XYZ fields [list "x" "y" "z"]
```

is executed at start-up. By default, vector types only support numerical indexed access, as in

```
atom get $ehandle $label A_XYZ(0)
```

which is equivalent to accessing the *x* value.

Bit-based data types, such as the *bitset* and *bitvector* types, will also allow the use of the property enumeration values as field names.

```
prop create E_MYBIT data type bit enum "^none:foo:bar"
```

will create a *bit* property, where the lowest two bits are named *foo* and *bar*.

```
ens set $ehandle E_MYBIT 1
puts [ens get $ehandle E_MYBIT]
puts [ens get $ehandle E_MYBIT(bar)]
ens set $ehandle E_MYBIT(bar) 1
puts [ens get $ehandle E_MYBIT]
puts [ens nget $ehandle E_MYBIT]
```

This sequence of commands demonstrates the use of enumeration values for bitsets. First, the property data instance on the ensemble is set to 1. The retrieval command will return *foo* as enumerated value, since only this bit is set. The direct check of the bar bit will report 0. When this bit is also set, the retrieval result will now be the list "*foo bar*", because now both bit 0 and 1 are set. The numerical value stored in the data slot is now 3, as demonstrated with the last line using the numerical retrieval command *nget* instead of *get*. *get* will use property enumeration values if they are specified, while *nget* outputs simple numbers if the underlying data type is numeric.

Subfield data types

The return value of

```
prop get A_XYZ fields
```

is a list, where every element is a nested list containing the field name and field data type, in the form of

```
{x float} {y float} {z float}
```

In case of vector types, the field data type is the same as the vector element data type which is already known to the toolkit and therefore should not be set explicitly to avoid conflicts. For compound properties, setting the field values is a requirement, though:

```
prop create E_MYDATA data type compound fields {{field1 int} {field2 blob}}
```

If the field value types are not set for compound types, they will default to strings.

Properties with polymorphic data types

Computation functions may be coded with introspection capabilities. This means, a computation function may look at the current data type of the property and provide data in the requested format.

This feature is actually used for some standard properties. Most of the image- and visualization properties can generate data either as file (data type *diskfile*), or kept in a memory block (data type *blob*). Examples of such properties are `D_GIF` (dataset depiction), `E_BARCODE` (barcode data as image), `E_EMF_IMAGE` (structure depiction in MS Windows vector formats) `E_EPS_IMAGE` (structure depiction in Encapsulated PostScript), `E_GIF` (structure depiction in pixel formats), `E_VRML` (VRML 3D models) and `X_GIF` (reaction depiction).

The default format of these properties depends on the toolkit version. Generally, full releases will use the *diskfile* data type, while special-purpose development libraries will use the *blob* type.

In scriptable toolkit versions, the data type may be changed by statements like

```
prop set E_GIF datatype blob
prop set E_GIF datatype diskfile
```

This should be done only once at the beginning at an application script, before any data is generated. It is possible to work with instances of both data types simultaneously, as will be explained in the next section, but this is usually very inconvenient and error-prone.

Changing property data types

In principle, it is possible to change the data type of a property after it has been defined. Chemical objects which hold property data in the old format remain linked to a temporary version of the old definition record. The old data type-specific functions are used to manage the property data, thus objects with old data can be safely deleted, written to file, etc. The old property definition record will be discarded only if there are no remaining data instances in the system which used the old definition. However, this old definition is only used for basic property maintenance and retrieval. All query functions on the data status will use the new definition and potentially return wrong results. On the scripting language level, the old definition is inaccessible.

When a data type is changed for a property which has a computation function, special care must be taken to either properly adapt the computation function to use the new data type, or to make sure that the computation function is never called - for example, by removing the computation function within the script as in

```
prop set $tmp_changed_property compfuncname {}
```

Reading files which were written with an outdated property definition in an updated environment can be problematic, especially if drastic changes such as a switch from a numeric type to a string type were performed. Especially binary formats are prone to crash the application after such a change. As far as the native **CACTVS** binary format is concerned (*.cbin* files), the following changes are safe, and everything else is a potential problem:

- Changing the external type without changes of the internal type, for example going from a *short* field to an *int* field.
- Changing the type when incidentally the internal formats have the same layout, for example going from *int* to *date*, or from *index* to *intquad*.
- Changing the standard size of any vector, matrix, or volume type. The old data will be read with the original size.
- Adding fields to *compound* properties, or reordering them. Deletion of fields is also possible, but only if the data types of the deleted fields are from the built-in set.
- Renaming enumeration values and other property attributes which only affect output formatting, but not the internal storage format.
- Changing field names, with the exception of *compound* properties, where these must not be changed in order to allow the later addition and reordering of fields.

The **CACTVS** scan/query file formats (both sequential and update-able) store definitions of all properties written to the file which are not in the built-in set, or were modified from the original definition. When these files are opened, the stored definitions are read back and supersede any previous definition in the application. An unfortunate side effect is that it is currently not possible to have two of these files open at the same time which contain conflicting property definitions.

The overall safest way to work with custom property definitions is to set them all up before the first chemical objects are created which rely on them, and to think carefully before defining them, so that no data sources with data encoded using conflicting definitions are encountered in a project.

Reactions

Reactions are major objects which control a collection of molecular ensembles as sub-objects. Their handle has the format **`reactionx`**, where x is a number. Any number of reactions may be created. As standard chemical objects, they may manage their own set of dataset-specific property data. The prefix for reaction data is “X_”. The prefix “R_” is already claimed by ring properties. The generic command for working with reactions is the *reaction* command.

Examples:

```
reaction set $xhandle X_IDENT "catalytic reduction"
reaction get $xhandle X_IDENT
```

Creating reactions

Reactions may be created by a number of methods. The supported mechanisms include:

- Decoding of string representations, for example **CACTVS** serialized reaction objects and Reaction SMILES:

  ```
  set xhandle [reaction create {CC=O.[H][H]>>CCO}]
  set xhandle [reaction unpack $packstring]
  ```

- Input from reaction files:

  ```
  set xhandle [molfile read "myreactionfile.rxn"]
  ```

  This method requires that the file contains reaction data, and that the read scope of the file input handle has been set to *reaction*. For most file formats which can store reactions this will happen automatically.

- Script-driven assembly:

  ```
  set xhandle [reaction create $reagent_ehandle $product_ehandle]
  reaction add $xhandle [list $solvent_ehandle "solvent"]
  ```

Internal structure of reactions

Reactions contain an arbitrary collection of ensembles as elements. The role of the ensemble is registered in the property `E_REACTION_ROLE`. The predefined roles are *reagent*, *product*, *solvent*, *catalyst*, *intermediate*, *impurity*, *byproduct*, *agent* (unspecified) and *undefined*. Additional roles can be created by editing the enumeration values of property `E_REACTION_ROLE`:

```
prop set E_REACTION_ROLE enum "[prop get E_REACTION_ROLE enum]:resin_material"
```

Usually, reactions contain at least a reagent and product ensemble, but this is not an absolute requirement. The order of the ensembles in the reaction is also arbitrary. One should not rely on the reagent ensemble being the first and the product ensemble being the second ensemble in a reaction.

It is not illegal to have multiple ensembles with the same reaction role within a single reaction, although for practical purposes this should be avoided.

Finding ensembles with a specific role in a reaction is best done by a filter on `E_REACTION_ROLE`:

```
set reagent_handle [reaction ens $xhandle reagent]
```

If there is no reagent ensemble in the reaction, an empty string is returned. An ensemble can only be a member of a single reaction, or not a member of any reaction. Ensemble membership in reactions is completely independent of membership in datasets.

The reaction handle can be obtained from an ensemble via a standard cross-referencing operation:

```
set xhandle [ens reaction $ehandle]
```

If the ensemble is not part of a reaction, an empty string is returned.

Reaction substructures

The toolkit supports reaction substructure searching, and SMIRKS transforms. Both require a reaction where the ensembles are not fully saturated with hydrogens. Such reactions may either be generated by specifying a decoder option, or automatically by decoding a Reaction SMARTS string on the fly in the query statement:

```
set rquery_handle [reaction create {[C:1]=[O:2]>>[C:1][O:2] smarts}]
molfile scan $filehandle "reaction >= $rquery_handle} reclist
molfile scan $filehandle {reaction >= [C:1]=[O:2]>>[C:1][O:2]} reclist
ens transform $ehandle {[C:1]=[O:2]>>[C:1][O:2]}
```

An important features for reaction substructure searches are atom maps. An atom map, which is stored in property `A_MAPPING`, links atoms on the reagent side of a reaction to atoms on the product side. In SMARTS/SMIRKS, these are specified by numbers (positive or zero) prefixed with a colon.

A reaction query "**C=O>>CO**" will match any reaction which contain a C=O group on the reagent side, and a C-O group on the product side. However, there is no requirement that there was any transformation of a keto or aldehyde group - this might simply be a match of a reaction structure which contains un-transformed keto and alcohol groups. Only by forcing the atoms to refer to a set of atoms linked by common mapping numbers actual reaction searching focusing on changed bonds can take place.

Reading and writing reactions

Reactions can be read and written to file formats which support the storage of reaction data. This includes for example MDL RXN files, MDL RD files, the native **CACTVS** binary file format (.cbin standard suffix) and the **CACTVS** query-optimized formats (.cbs standard suffix).

Writing to a format which supports reaction storage is simply achieved by passing a reaction handle as write object. Reaction handles may also be passed to output channels configured to formats which do not support reaction storage, but in this case the reaction is split into individual ensembles and multiple records are output. Example:

```
molfile write myreaction.rxn [reaction create C=O>>CO]
molfile write mydata.sdf [reaction create C=O>>CO]
```

The first example will write a reaction record, while the second example will write two SD file records.

When reading data from reaction files, the read scope parameter of the structure file object must be set correctly. Only if this parameter is set to *reaction* (and not *mol*, *ens*, or *dataset*), reaction objects will be read. For most file formats, the scope parameter is automatically set to the most complex data object contained in the file. For example, RXN files are automatically opened for reaction input, and

SD files for ensemble input. In most cases, reaction files can also be read on an ensemble level. An RXN file opened with read scope *ens* or *mol* will appear to contain two records:

```
set fhandle [molfile open mydata.rxn r readscope ens]
set enslist [molfile read $fhandle {} all]
```

The ensemble list will contain two ensembles. The file could also be read with two individual single-record **`molfile read`** commands. Reaction-level information is of course lost when files are read this way.

Both RD files and simple **`Cactvs`** binary files should be opened with explicit read scopes. The situation is additionally complicated by the fact that both formats could conceivably contain a mixture of structure and reaction records.

Datasets

Datasets are major objects used to organize collections of ensembles or reactions. Datasets have handles in the form **`dataset`***x*, where *x* is a number. Any number of datasets may be created. As standard chemical objects, they may manage their own set of dataset-specific property data. The prefix for dataset properties is “D_”. The generic command for working with datasets is the *dataset* command.

Example:

```
dataset set $dhandle D_NAME
dataset get $dhandle D_SIZE
dataset get $dhandle D_GIF
```

The example code sets a dataset name, queries the size (number of elements) of the dataset, and finally generates a panel compound image, where the panel grid is filled with images of the dataset objects.

Compatibility features

For historical reasons, the alternative name *queue* for datasets is still supported. This extends even to the decoding of handles - handle *queue0* is equivalent to handle *dataset0*.

Again for historical reasons, a standard empty dataset *dataset0* is automatically created when the toolkit is initialized. This dataset cannot be deleted.

Elements of datasets

Datasets can contain an arbitrary number of ensembles and/or reactions as elements as an ordered sequence of objects. Datasets which mix ensembles and reactions are possible, but rarely useful. Currently, an ensemble or reaction can only be a member of a single dataset at any time, or be not part of any dataset. By default, ensembles or reactions are created or read without being a dataset member. Ensembles or reactions can be moved between datasets. Many input and creation commands have optional parameters to identify a target dataset and will deposit the newly created objects directly into it.

Examples:

```
set dhandle [dataset create]
```

```
ens create CCC 1 $dhandle
molfile read "z.sdf" $dhandle
ens move $ehandle $dhandle
ens move $ehandle {}
```

The example sequence shows how to create a dataset, generate an ensemble in a dataset, read a file directly into a dataset, move an existing ensemble into a dataset and then to remove it from the dataset again.

Dataset file I/O

When the native Cactvs binary format is used, dataset property data can be stored and retrieved. Unfortunately, standard chemical exchange formats do not support the storage of global dataset-level information. The command sequence

```
molfile write "dataset.cbin" $dhandle1
set fh [molfile open dataset.cbin r readscope dataset]
set dhandle2 [molfile read $fh]
molfile close $fh
```

first writes a dataset identified by the handle *$dhandle1* to the file *dataset.cbin*. By using the standard suffix *.cbin*, the file is automatically set up to use the native CACTVS binary data format without the need to explicitly set the file format. The *write* command is supplied with a dataset handle as parameter. For output to file formats which support dataset-level information storage, this implies that the dataset-level property data should also be stored, together with the dataset elements.

When the file is later opened for reading, it must be specified that dataset-level input is requested. The *read* command will then return the handle of a newly created dataset, which has the dataset property data from the file, and in addition contains newly created instances of all the original elements in the dataset (ensembles or reactions) with their original data. If the read scope is not adjusted to *dataset* prior to reading, dataset files can still be read as normal structure or reaction files, returning ensemble or reaction handles as the result of *read* commands, but the dataset-level information is ignored.

If the file in the example were a simple SD file, the write statement would write a set of records for the elements, and the read statement return a dataset with all the original elements, but without the dataset-level information.

Virtual datasets

The *dataset* command does not actually need to be used with a proper dataset handle as identifier. The handle parameter may be replaced by a list consisting of any combination of dataset handles, ensemble handles, and reaction handles. If the parameter is anything but a single dataset handle, all specified objects are temporarily moved into a virtual dataset. In case dataset handles are part of the list, the objects contained in the listed dataset are moved into the temporary dataset, not the dataset itself.

When the command has finished, the objects are moved back to their original datasets, in their old position, or reset to their original dataset-less status. A few logical exceptions apply to this rule. For example, the global move command *dataset move* will of course let the moved objects remain in their new destination and not pop them back into their old place as soon as the command finishes.

Examples:

```
dataset move [list $ehandle1 $ehandle2 $ehandle3] $dhandle
dataset scan [list $dhandle $ehandle1 $ehandle2] "structure >= c1ccccc1" enslist
dataset get [dataset list] D_SIZE
```

The first example will append the three ensembles to the specified dataset. The second example performs a substructure search on the combined list of all the elements in the argument dataset and the two additional ensembles. The third example retrieves the total element count of all datasets.

It is possible to set dataset properties on virtual datasets, but of course this information is immediately lost when the virtual dataset is destroyed.

Because a ensemble or reaction can only be a member of a single dataset, multiple listings of the same ensemble or reaction in a virtual dataset list are ignored. The processed virtual dataset contains these objects only once, at the position of their first listing.

Structure, reaction and dataset file I/O

The toolkit has an extensive system for intelligent structure and reaction I/O. The central object used for this purpose is the **`molfile`** major object. A **`molfile`** object does not necessarily refer to a file in MDL Molfile format, but can be used to access any identified structure file format.

Opening and closing structure files

Different from other major objects, **`molfile`** objects always refer to one or more files on the file system or an in-memory string representation of a file. They not created with null data, but always refer to some collection of file data. A **`molfile`** object is identified by its object handle. The object remains active until it is closed. After that, the handle becomes invalid, but in most cases, the file it referred to is preserved.

```
set fh [molfile open "myfile.sdf"]
molfile close $fh
```

The **`molfile close`** command can close all open structure files by passing the special handle *all*.

```
molfile close all
```

File modes

Molfile objects can be opened in different modes. The default mode is *r*, meaning the file is opened for input at the beginning of the file.

```
set fh [molfile open "myfile.sdf" r]
```

The statement above is completely equivalent to the statement five lines above. Other important modes are *w* for overwriting and *a* for appending to the end of a file. File in formats which can be rewritten on a per-record level without disturbing records behind the rewrite position can also be opened for update with mode *u*.

```
set fh [molfile open "myfile.sdf" w format sdf]
molfile write $fh [ens create CCC]
molfile close $fh
```

The three statements above will write a single MDL SD-file record with the propane molecule to file *myfile.sdf*. If the file existed before, it will be overwritten. If the file had been opened with mode *a*, the record would have been appended. Opening a file which does not yet exist in mode *a* is equivalent to opening it in mode *w*.

Input file formats and I/O modules

For input, the format of a file is autodetected. This feature works by looking at data at the beginning of the file. In case the file cannot be rewound for later reading, the bytes needed for peeking are internally buffered. Generally, it is not a good idea to specify the file format for input directly. The format of a file is *not* determined by looking at the suffix of the filename, but there is a twist explained below.

File I/O in the **CACTVS** toolkit is extensible by loading I/O modules. Very few I/O formats are built-in. In standard configuration of the basic interpreter, these are only the native formats of the toolkit, **SMILES**, a couple of MDL formats (**SDF**, **RXN**, **RDF**), and meta formats (*mailbox*, *hitlist*). Other interpreters may have an extended set of compiled-in I/O modules. Nevertheless, most standard interpreters will load a couple of additional I/O modules at start-up, and these can be used in the same fashion as built-in modules. The list of auto-loaded I/O modules can be configured in full toolkit installations by editing the *siteconfig/cactvsio* file.

When a file is opened for reading, the format detection modules of all loaded I/O modules are invoked, with the most recently loaded module first and the built-in modules last. If any of the format detection routines claims it has detected the format, the issue is considered settled and the selected I/O module will handle all further I/O to and from this file.

If the file format identification failed, the toolkit will make an attempt to auto-load a suitable I/O module, if the interpreter has support for loading of dynamic modules. Only in this case is the suffix taken into account.

```
set fh [molfile open "result.pdb"]
```

If the interpreter executing the above statement does not have built-in support for **PDB** files, and the **PDB** I/O module is not yet loaded, the interpreter will try to locate the module along its search path, which can be configured in global variable **cactvs(filexpath)**. If the interpreter can find the module named *filex_pdb.so* or *filex_pdb.dll* (following the platform-dependent naming conventions of shared libraries, and using the filename suffix as part of the module name), this module will be automatically loaded, and its format detection routine given a chance to identify the format. However, if a statement like

```
set fh [molfile open "result"]
```

fails, because the file *result* is of a format which cannot be identified by the built-in and currently loaded modules, explicit action needs to be taken. In such cases, explicit loading of a suitable file format can be requested:

```
filex load pdb
```

Above statement will load (or reload) the **PDB** I/O module explicitly. Automatic loading of I/O modules is an incentive to use standard suffixes. However, as long as the proper I/O modules are loaded, omitting suffixes or even using misleading suffixes is no problem.

Output file formats

The default format for output files is determined by its file name suffix. If no suffix is provided, the system default format (usually **SDF**) is used.

If the suffix is not associated with a loaded I/O module, the I/O module search path is traversed and the I/O module automatically loaded, if it can be located on the path, and the interpreter is allowed to load extensions.

```
set fh [molfile open "myresult.pdb" w]
```

Above statement will open a **PDB** file for writing, if the **PDB** module is loaded, or can be auto-loaded.

However, for output is is customary to specify the format explicitly, if just for resolving suffix use conflicts. For example, the suffix *.mol* is not specific for any single format, and the actual format used depends on which modules are currently loaded.

```
set fh [molfile open "myresult.mol w format sybyl2]
```

In contrast, above statement is unambiguous. The **Sybyl2** I/O module is automatically loaded if required and found on the search path. Alternatively, it could be explicitly loaded before executing the file open statement via a

```
filex load sybyl2
```

statement, or, if the module is located in a location outside the standard path, with an extended command like

```
filex load sybyl2 /private/modules/filex_sybyl2.so
```

File attributes

Molfile objects have a complex inner structure, and consequently a lot of attributes which can be queried and set. File attributes can be set by the **molfile open** statement, or at any later time set and queried with the **molfile set** and **molfile get** commands. The **format pdb** and **format sybyl2** parts of the **molfile open** statements used in the sample statements in this section are already examples of file attributes set at the time of opening the file.

```
set fh1 [molfile open infile.pdb r hydrogens add]
set fh2 [molfile open outfile.sdf w format sdf subformat 2D writelist E_WEIGHT]
```

These statements show some typical attributes settings. After the **hydrogens add** attribute has been set, a standard set of hydrogens will be automatically added to any structure or other object read from the input file. The **subformat 2D** attribute configures the output file to explicitly use 2D coordinates. They will be computed if not yet present. By default coordinates already present will have precedence, so if the structures to be written to this file are read from a `PDB` file with atomic 3D coordinates, the written SD file records will have 3D coordinates. The **writelist** attribute configures a list of properties to be written as SD data fields. Here, we add molecular weight. The default data field list is empty, so no SD data fields are written.

Field attributes can also be queried:

```
set fmt [molfile get $fh format]
set rec [molfile get $fh record]
set line [molfile get $fh line]
```

These sample commands show how to retrieve the file format, the record number (beginning with 1) of the next record to be read or written, and the current line number on the file. With the exception of the **`line`** attribute, these fields can also be set:

```
set fh [molfile open datafile.sdf]
molfile set $fh record 5
set eh [molfile read $fh]
```

With this command, the structure read with the **`molfile read`** command will be the one in the fifth SDF record.

The most often used file attributes are:

• **`format`**	The file format. Usually set only for output. Useful to query on input files to learn the actual file format as detected.
• **`record`**	File record of the next record to be read or written, starting with 1. For normal files, the record position can be set to arbitrary input locations anywhere in the file, regardless whether the location has been visited before or not. In files which cannot be rewound, only forward skipping is possible. It is possible to set the record position on output files, too. However, if the file cannot be rewritten in place, or the file mode is not *u*, all data behind the new location is deleted.
• **`line`**	The current line number in the file. After a record has been read, it is the number of the last line of the record. Binary files count one line per record. This attribute can be set for bookkeeping purposes, but does not change the actual file position. A newly opened file reports a line number of 0.
• **`eof`**	This attribute is set to 1 if the file reached EOF on input. It cannot be set.
• **`hydrogens`**	The hydrogen addition mode. The default is *asis*, meaning that no hydrogen manipulation takes place on input and output. Other important modes are *add* (add standard set), *strip* (strip all hydrogens except those usually shown in structure drawings) and *stripall* (remove all hydrogens). Note that there are actually two different hydrogen addition modes, one for output and one for input. The changed or reported attribute is the one belonging to the current file mode (which could be changed with a **`molfile toggle`** command).
• **`eolchars`**	The end-of-line characters used for output of ASCII files. The default is dependend on the platform (*\n* on Unix/Linux, *\r* on MacOSX, *\r\n* on Windows). This attribute does not give information on the type of EOL character used on input files.
• **`fields`**	The names and possibly data types and other attributes of data fields. The exact meaning of this attribute depends on the file format. For SD files, it is a list of the original names of the SD data fields. For file formats which can have different data content on every record, such as SD and RD files, the attribute is updated after each input record, and it is empty before data has been read. For file formats with a header detailing the data structure, it is filled when the file is opened. With the exception of the **`CBS`** and **`BDB`** file formats, this attribute is intended to be used for input files only. Setting this attribute does not have any effect

on output to, for example, an **SD** or **RD** file. Please use the *writelist* attribute for this function. For **CBS** and **BDB** files, setting this attribute changes the global layout of the file. This is explained in more detail in the chapter about database files.

- **readflags** This attribute is a list of flags which control various aspects of structure processing during and immediately after reading, before the handle of the read object is returned as input result. The most important flags from a rather extensive list are:
aroresolver: If this flag is set, bond which are marked as aromatic but do not have a defined Kekulé form are automatically resolved into a Kekulé structure. A common use for this feature is for reading MDL Molfile with type 4 bonds, which are only allowed to be used as query bonds, but are frequently found in registration system data too.
complexresolver: Change the bond type of bonds which cannot be reasonably represented as standard VB bonds into *complex* bonds, which are exempt from bond electron counting. This is the only flag which is set by default.
fixstereo: Discard wedge bonds on atoms which cannot be stereocenters. This option requires that hydrogens are added during the read (**hydrogens add** attribute).
mergedata: If a file contains multiple instances of a data field, by default multiple instances of the associated property are attached to the ensemble or other input objects. For example, an SD file record with two <*mydata*> fields will produce an ensemble with properties `E_*MYDATA*` and `E*MYDATA*/2`. If this flag is set, the data of repeated fields is appended to the first property instance. The exact meaning of *appended* is dependent on the datatype of the property. For the common case of strings, it is appended at the end, separated by a tab. For vector types, additional elements are created.
noeof: Ignore EOF indications on the input channel.
noimplicith: Do not assume the file contains implicit hydrogens. With this option, **SMILES** data is read in a pseudo-**SMARTS** fashion, without added hydrogens. This option has no effect on file formats where atoms do not possess an implicit hydrogen count, such as **MDL Molfiles**.
nometal: If this flag is set, assume that the file does not contain metals, Symbols which look like the element names of metals are instead interpreted as superatoms, for example *Al* for alanine.

- **readscope** This attribute controls the type of object read from an input file. The default value depends on the format of the input file. For most formats, it is *ens*, i.e. the object read is an ensemble. For **RD** files, **RXN** files, and reaction **SMILES** the default format is *reaction*. Files with a data description heade are automatically opened with a read scope matching the file content (i.e. a **BDB** reaction file is opened in reaction scope, while a structure file is opened for structure input). The allowed values of this attribute depend on the file format. For example, an **RXN** file can be read in mode *ens*, retrieving one half of a reaction per read. Other modes which are allowed for specific formats are *mol* (reading individual molecules from ensembles and returning them as isolated ensemble objects) or *dataset* (retrieving complete datasets as dataset objects with structure or reaction subobjects, including dataset-level property data).

- **`subformat`** This attribute identifies various format variants of certain file formats. It can be both read, after retrieving a file record, or set to influence the formatting of an output file. The must important subformats are mol*2D*, mol*3D*, and mol*0D* which can be used to identify or control the type of coordinate data in **`MDL Molfile`** variants (**`Molfile`**, **`SDF`**, **`RXN`**, **`RDF`**).
- **`writeflags`** This attribute is a list of flags which control various aspects of file output. The most commonly used flag bits, which are specified as a list of set flags, are:
  compute: Attempt to compute properties on in the *writelist* attribute. By default, they are only written if they are already valid at the moment of writing. No error is raised if computation for a listed property fails.
  nostereo: Do not output stereo information, even if it is present. This is for example useful in case a 3D compound with arbitrary, but in reality undefined stereochemistry is write as a 2D structure.
  write0D: Write a 0D record without coordinate information, if the file format supports this.
  write2D: Write a 2D record with atom display information, if the file format supports this.
  write3D: Write a 3D record with atomic 3D coordinatges, if the file format supports this. The effect of the *writexD* flags is the same as setting the corresponding file subformat.
  writearo: Write aromatic bonds with an aromatic bond type, even if this is questionable, such as in MDL Molfiles which should encode a defined Kekulé structure.
  writename: Add the contents of property `E_NAME` as a name field if this is optional, for example when outputing **`SMILES`**.
- **`writelist`** This is a list of properties which should be output as additional data fields, if the file format supports this. However, by default no attempt is made to actually compute this data. If a property is not present, it is, dependent on the file format, silently ignored or stored as a **`NULL`** value. Computation can be automatically attempted for all listed properties by setting the *compute* flag of the *writeflags* attribute. Properties which cannot be output in a file because of restrictions of the file format with respect to supported datatype or property object associations are also silently ignored. The default *writelist* is empty, meaning that no extra data beyond the minimum defined by the file format is output by default.
- **`droplist`** This is a list of properties which should not be output, even if present and listed in the *writelist*. It effects only file output operations.
- **`ignorelist`** This is a property list which affects input only. All properties listed here are ignored and deleted from the read structure object, even if the input record explictly contained this data.

This is an example script showing some common uses of file attributes:

```
prop set E_WEIGHT origname “Molweight”
set fhin [molfile open “infile.sdf” r hydrogens add]
set fhout [molfile open “outfile.sdf” w format sdf writeflags compute \
   droplist “Activity” subformat 2D
set eh [molfile read $fhin]
molfile set $fhout writelist [concat [molfile get $fhin fields] E_WEIGHT]
molfile write $fhout $eh
molfile close all
```

This script converts a (presumably) 3D molecule in file *infile.sdf* into a 2D record in file *outfile.sdf.* All SD data fields in the input file are copied to the output file, except for the field *Activity*. This is done by copying the field information from the input file, extracted after reading the current input record, into the *writelist* of the output file, but listing the Acitivty field in the *droplist*. An additional field for the molecular weight is added, under the field label *Molweight*, and it is computed if necessary.

One-shot file command shortcuts

Most **`molfile`** subcommands accept, in addition to a file handle obtained from from a **`molfile open`** statement, the name of an existing file. If the resolution of a file handle fails, an attempt is made to open the identifier as a file for reading, except if the command is an output operation. The **`write`** subcommand opens the file in *w* (overwrite) mode, while the **`delete`**, **`reorg`**, **`rewrite`** and **`set`** commands open in *a* (append) mode. If this operation succeeds, a temporary handle is created, which is automatically released when the command finishes.

Examples:

`set eh [molfile read myfile.skc]`

```
set nrecs [molfile count myfile.sdf]
```

These commands are identical to the sequence

```
set fh [molfile open myfile.skc]
set eh [molfile read $fh]
molfile close $fh
set fh [molfile open myfile.sdf]
set nrecs [molfile count $fh]
molfile close $fh
```

Here is a very simple output operation:

```
molfile write out.sdf [ens create c1ccccc1]
```

This command creates a single-record **`MDL Molfile`** in the current directory, overwriting any existing file of that name.

Reading structure objects from files

The primary input command is **`molfile read`**. It reads the next record from an input file identified via its file handle and returns an object which corresponds to the current read scope of the file handle. In most cases, this is an ensemble object. The input objects are allocated and should be freed if they are no longer needed. Here are some sample commands:

```
set eh [molfile read “mymol.sdf”]

set xh [molfile read [molfile open “reactions.rdf” r record 2 hydrogens add]]

molfile read “bigfile.sdf” [dataset create] all
```

The first line simply tries to open the file *mymol.sdf* and reads the first record, without performing any kind of modification or standardization on the data.

The second example opens a reaction file and positions the read pointer to the second record. In addition, any object read from the file gets a standard set of hydrogens added. Assuming that the RD file contains reaction information, and the read scope was thus automatically set to *reaction*, the outer **`molfile read`** command returns a reaction object.

The third sample line reads the complete file into a dataset object. The optional third parameter of the **`molfile read`** command is a recipient object handle. If, for example, the recipient object is a dataset, and the objects read from the file are ensembles or reactions, these objects are appended to the dataset object. It is also possible to specify an ensemble object (if the read scope is *ens*), or a reaction object (if the read scope is *reaction*). In that case, the existing object is cleared, but its handle reused for the new data. This can be useful in case there are references to an exisiting ensemble object which are difficult to update. Instead of omitting this parameter, an empty string may be used.

The optional fourth parameter of **`molfile read`** is a record count. Its default value is one. The special value *all* can be used to read until the end of the file. In case an explicit record count is used, the command returns the list of successfully read object as object handles, even if their number is less than the requested number, provided that at least one object could be read. Otherwise, an error is raised. The distinction between reaching EOF and encountering an input error is easily made with code such as

```
if {[catch {molfile read $fh} eh]} {
   if {[molfile get $fh eof]} {
      # normal eof
   } else {
      # input error
   }
}
```

The **`molfile hread`** command has the same syntax as **`molfile read`**, but it adds a standard set of hydrogens to the read objects, without permanently changing the hydrogen addition attribute of the file. Example:

```
set elist [molfile hread "myfile.sdf" {} 5]
```

The code above reads the first five structure records from the file and adds hydrogens to the ensembles. The return value is a list of five ensemble handles. If the file contained only a smaller number of records, but at least one record, a shorter object list is returned.

Looping over files

Processing an input structure file from beginning to end is a very common task. It can be done with code such as this:

```
set fh [molfile open "bigfile.sdf" r]
while {![catch {molfile read $fh} eh]} {
   # process structure here ...
   # now get rid of input object
   ens delete $eh
}
if {![molfile get $fh eof]} {
   # process error
}
molfile close $fh
```

Since this is somewhat cumbersome, and due to the commonality of the task, the toolkit has a convenience function **`molfile loop`**. The sequence above can be simplified to

```
molfile loop "bigfile.sdf" eh {
   # process structure
}
```

The first two mandatory parameters of the **`molfile loop`** command are the file handle (or a handle temporarily generated for a one-shot file, as in the example above) and the name of a **Tcl** variable which will hold the handles of the read objects for use within the body of the function. The last (but not always third) parameter is the function body. Any number of scripting commands can be put there.The standard **Tcl** commands **`break`**, **`continue`** , **`return`** and **`error`** will work as expected within the loop body, just as they work in a **`for`** or **`while`** loop. It is possible to loop over reactions or datasets if the read scope is set appropriately. The object handle of the current read item is stored in the variable regardless of its type.

Note that the loop example did not delete the ensemble read from the file. This is done automatically when the loop body was executed and the loop is prepared to execute the next cycle. Any explicit deletion of the input object within the loop is silently ignored. The object is undeletable until the loop body has finished.

The loop finishes silently on EOF. In case an input error is encountered, an error is raised. Thus, no check for distinguishing EOF from a read error is required. If a loop throws an error, the file is damaged.

The loop begins at the current record. If the file is not positioned on the first record, it is not rewound.

The maximum number of iterations of the loop can be limited by an optional iteration count parameter inserted between the storage variable parameter and the function body. A negative value or an empty string indicates an endless loop.

The command variation **`molfile hloop`** takes the same parameters as **`molfile loop`**, but automatically adds a standard hydrogen set to the input objects. Just like the **`molfile hread`** command, it does not permanently change the hydrogen addition attribute of the file.

An example demonstrating a few of the features mentioned in the last paragraphs:

```
set xh_aldol ""
if {[catch {
   molfile hloop reactions.rdf xh 10 {
      if {[reaction get $xh X_NAME] = "Aldol condensation"} {
          set xh_aldol [reaction dup $xh]
          break
      }
   }
} msg]} {
   puts "read error: $msg"
}
if {$xh_aldol!=""} {
   # do something
}
```

This script searches the first ten records of reaction data file *reactions.rdf* for a reaction with the name “Aldol condensation”. Assuming that the RD file contains reactions, the input objects generated by the loop are reactions with fully specified hydrogens. Regardless how the loop is left, the loop input object is automatically deleted. So in case you want to export an object from within the loop, it must be duplicated. The **`break`** command exits the loop when the desired reaction was found. The outer catch command wraps the full loop statement. It is triggered when a read error occurred, but not if EOF was found, ten records were examined, or the loop left via the **`break`** command. An error message is captured in the **`msg`** variable and printed if the **`catch`** command triggers.

The execution of **`molfile`** loops can be made more robust with two special file attributes which are only checked within these loops. These *readflags* attributes flags are supported:

• *ignoreempty*	Empty input records (i.e. ensembles or reactions without atoms) are silently skipped.
• *ignoreerrors*	If a read error occurs, the defective record is ignored and the loop tries to resynchronize and continue with the next record. An error is raised only if the resynchronization fails.

The return value of the **`molfile`** loop commands is the number of iterations successfully executed. Ignored faulty records are not counted.

Example:

```
set fh [molfile open “big_damaged_file.sdf” r \
   readflags [list aroresolver ignoreempty ignoreerrors]]
set loopcnt [molfile loop $fh eh {
      # process, ignore problems as much as possible
}]
puts “$loopcount records successfully processed”
```

Command extensions and modules

The **Cactvs** toolkit includes a collection of modules which provide additional functionality when loaded. Some library versions may also contain compiled-in versions of these modules.

Tcl and Tk Packages

All Tcl- and Tk-enabled toolkit versions may load standard Tcl extension packages via the standard Tcl mechanisms. The Tk toolkit is itself available as a package any may be loaded into any plain Tcl-enabled script interpreter. The **`package require`** command will take care of package dependencies and automatically load additional packages if required and hide platform specifics such as the suffix of dynamic link libraries, shared libraries, or bundles.

Examples:

```
package require Gd

package require Gdbm

package require Tk

load $cactvs(libdir)/libTktable2.8.so
```

Packages usually provide at least one additional command to the interpreter they were loaded into. By convention, the name of this command is the same as the package name, but spelled in all lower case.

Example:

```
package require Gd
set gdhandle [gd create 100 100]
gd rectangle $gdhandle [gd color new 255 0 0] 0 0 50 50
```

Packages are automatically located in all directories which are listed in the standard Tcl variable **`tcl_pkgPath`**. This variable may be changed as needed.

Loading a package more than once does not have any effect. Loaded packages are only accessible in the Tcl interpreter which loaded them. For example, a Tcl interpreter associated with a Tcl computation function does not have access to the commands of a package loaded by the main interpreter, if the main interpreter did not explicitly export the commands to the slave.

CACTVS Command Extensions

CACTVS command extensions are very similar to Tcl packages. They may even be loaded as such vie the standard Tcl **`package`** or **`load`** commands. The only major difference is that they contain a table with module information in addition to the standard Tcl module initialization functions. If command extensions are loaded via the **`cmdx load`** command, this data is accessible and may be queried in scripts.

Example:

```
cmdx load stat
puts "Version: [cmdx get stat version] by [cmdx get stat author]"
puts "correlation coefficient: [stat r {1 2 3} {5 6 8}]"
```

Command extensions are located automatically in all file system directories and other places, such as databases or Web locations, listed in the control variable **`cactvs(cmdxpath)`**. This path variable may be changed as needed.

Command extensions are global. Once a command extension has been loaded, it is usable in all slave interpreters, such as computation interpreters associated with properties.

The Gdbm Module

The Gdbm module is a standard Tcl module. It provides a high-level access to the Gnu Gdbm library. Because this library is under the GPL (not even LGPL) license, it is only part of selected distributions.

The module is usually loaded with a **`package require`** command:

```
package require Gdbm
```

The purpose of the module is working with Gdbm files. Gdbm files are simple keyword/value storage files with an efficient, hash-based random access mechanism via the keyword, which can be any string. In principle, these files are comparable to Tcl array variables - but since the content is held on file, they need far less memory resources.

A common application example for these files is the storage of large structure collections, without the overhead of a real database, or a scan file. Structure hashcodes (`E_HASHY`, `E_HASHSY`, `E_HASHTY`, etc.) are useful access keys - but any identifier, such as a record number, `E_IDENT` ID string, etc. works as well. The value part of an entry may be anything, from a complete structure information (conveniently packaged as pack string - see **`ens pack`** command), via a SMILES string (property `E_SMILES`) to any kind of other information, such as a compound name.

Examples:

```
package require Gdbm
set ghandle [gdbm new test.gdb]
set eh [ens create COC]
set hash [ens get $eh E_HASHY]
gdbm insert $ghandle $hash [ens pack $ehandle]
if {[gdbm exists $ghandle $hash]} {
   set enew [ens unpack [gdbm get $hash]]
}
gdbm close $ghandle
```

The access to an opened Gdbm file is performed via a handle. Commands which open or create a Gdbm file will return a handle, which should be saved and passed to subsequent commands as the file unidentified. The mechanism is essentially the same as for the toolkit chemistry objects, such as ensembles, reactions, or structure files. Handles remain valid until the file is closed.

Gdbm files are platform dependent and cannot be opened on a computer with a different byte ordering. Exchanging these files between e.g. IRIX and Linux computers is not possible. This limitation is inherent to the file format and not a problem of the toolkit.

The Gdbm module commands are detailed in the reference section.

SQL Expressions

The **CACTVS** toolkit contains, if the compilation flag for this feature was set, an SQL-compatible function parser which is used in various contexts. Its most important applications are:

- Function columns in table objects
- Row selection in table objects
- General support for free-form data formatting for chemical objects

The parser is used to evaluate expressions which usually involve property data. The mechanism of referring to property data is dependent on the context and will be described in more detail below.

Function Syntax

The function syntax follows mostly the expected schemes. Normal operators obey the same precedence rules as in C. Parentheses may be used to group expressions.

Examples:

```
1+2*3
```

```
concat("a","b","c")
```

In contrast to standard SQL, case does matter. All function names must be written in lower case. Property references are written as uppercase strings.

Example:

```
interval(E_WEIGHT,100,250)
```

An unusual aspect of the function syntax is that some syntax elements use embedded keywords. In addition to the function names, these keywords are reserved. Keywords must be separated by whitespace or punctuation characters from the rest of the expression.

Examples:

```
substring("abc" from 1 for 2)
3 in(1,2,3)
(1/0) is NULL
```

At every stage during the evaluation of an expression, a check is made whether any of the input parameters for the next step in **`NULL`**. If it is, in most cases the full sub-expression will also become **`NULL`**. The meaning of **`NULL`** is not an empty string, or a **`NULL`** pointer. Rather, it indicates unspecified or undefined data. Making decisions with undefined data is not possible. A check whether **`NULL`** equals **`NULL`** will also result in **`NULL`**: Since both comparison values could be unspecified in different ways, it is not possible to obtain a valid comparison result. However, there are a few functions which allow the explicit test for **`NULL`** values and reacting towards it.

`NULL` values may for example be encountered by referring to unset table cells if the SQL expressions are used in a table context.

Example:

```
ifnull(1/0,9999)
```

Data types in expressions and functions

The result of a function can only be a signed integer, a double precision floating point value, a date value, or a string. The result data type is usually determined by the functions use in the expression. For mathematical expressions, the arguments are automatically adapted. If any element of a simple mathematical expression is a floating point number, the result will be a float. If a mathematical expression involves string parameters, an attempt will be made to interpret the strings first as an integer, and then a float if the string is not a simple integer. An initial integer value may again be promoted to a float in case it is used in a floating point context. Date values used in a numerical context will be treated as an integer (internally, dates are stored as seconds since Jan 1st, 1970).

Examples:

```
1+2.0
```

will yield a floating-point **`3.0`** as result, while

```
1+"2"
```

will yield an integer result of 3.

Numerical values which are passed to functions which expect string input parameters are formatted as standard integers or *%g* floating point values and then passed as a string.

Property values which are used as function or expression arguments are cast to the expected type. If the property is of a data type which does not provide a suitable cast function to any of the allowed types, an error results. The use of property subfields as function or expression arguments is supported.

Example:

```
set ehandle [ens create {C methane 108-88-3}]
puts [ens expr $ehandle {concat('CAS# ',E_NAME(1),' ',E_WEIGHT)}]
```

This example first generates an ensemble, using the feature to transfer naming information as part of the SMILES string. The name, which here is actually a composite of two parts, is automatically stored as property `E_NAME` as part of the ensemble information. Access to the second word of the name is possible by means of standard word-based indexing on string data (indices start with 0), and that data is then concatenated with the leading constant string. A separator space is added, and then the floating point property value `E_WEIGHT` of the ensemble is cast to a string (after it was computed on the fly) and appended. The final output of the command is "*CAS# 108-88-3 16.0426*".

Using expressions which check for data availability, it is possible to output or import different property data into a column depending on the circumstances.

Example:

```
table addcol $thandle function "ifnull(E_MDLNUMBER,E_COMPANY_IDENT)" molid
```

This example adds a function column to a table. The name of the column is *molid*. When an ensemble is later added to the table, it is filled with data from property `E_MDLNUMBER`, if is was present or could be computed. Otherwise, data from the second property `E_COMPANY_IDENT`, which could be an in-house identifier, is used. If that identifier is also missing, the table cell receives a **`NULL`** value.

Names of properties or fields/columns may be dynamically constructed by casting a string to a property or field/column reference:

```
table sqlselect $thandle {property(concat("E_","WEIGHT")) between 100 and 250}
```

This example will select all table rows from a table where the ensemble molecular weight of the entries lies between 100 and 250. If a table column with data of property `E_WEIGHT` exists[10], it is used for the scan. Otherwise, a check will be made whether the table rows are associated with ensembles which are still in memory. If that is the case, the weights will be retrieved or calculated from these ensembles. If the ensembles do not exist any longer, or the data was stored without leaving ensemble references, the selection function will see **`NULL`** values.

Another application area for SQL expressions is data formatting via the *expr* command of chemical objects.

For standard ensemble data formatting, it is usually not necessary to use the SQL parser. The standard formatting capabilities built into Tcl will work as least as well. These expressions develop more power when used in the context of table function columns and selection functions.

10. This is independent of the column name - we are looking for a column of type *property* which is linked to property **`E_WEIGHT`**

Function references to constants and data

These types of data can be used in SQL expressions:

• Integer constants	Standard signed numbers. Example: 999
• Hexadecimal constants	These start with a 0x character pair and continue with a sequence of case-insensitive hex digits (0-9 and a-f). Octal constants are not supported. Example: 0xff
• Floating-point constants	Floating point constants must contain a decimal point, and may use an exponent. Usually, it is not necessary to specify integral floating point values as such, because integers are cast automatically to floats when required. Examples: **`2., 3.14159, 2.97e6`**
• String constants	Strings are started by either a single or double quote, and extend until a closing quote of the same type is found which is not escaped. The escape character is a backslash. The maximum length of a string is currently 8K. Three-digit octal escape sequences as well as the standard escapes ’\n’, ´\t’, ’\r’,’\b’, ’\f’, ’\v’ and ‘\z’ are recognized and decoded. Multi-line specification of strings with a backslash as last character on a line is also supported. Examples: **`"Hello World", '"We\tare\tthe\tchampions\n"'`**
• **`NULL`** or **`null`**	The function evaluation engine is fully **`NULL`**-compliant according to the standard. Constant **`NULL`** values are supported.
• **`true`** or **`false`**	These are just alternative names for integer constants 1 and 0.
• Uppercase property name	Reference to a property attached to the context object. The context object can for example be an ensemble. The expression context must match the property object class - it is for example not possible to refer to an atom property in an ensemble context. Only native **CACTVS** property names may be used, not original names as they were read from a data file. In order to be able to use property data, the chemical object providing the data must still be around. In the context of tables this means that an update of function cells referring to properties is not possible if the original ensemble or reaction which provided the data is no longer present and (see below) there is no data column which holds a copy of the original ensemble or reaction data. Example: **`E_NAME`**

• Uppercase indexed property	Properties may be indexed. In contrast to the standard property indexing mechanism in the scripting environment, this index may be a dynamically evaluated SQL function. If the index is a constant field name string and not a numerical index, it must be quoted as a string, which is an important difference to the standard script indexing syntax. Only native **CACTVS** property names may be used, not original property names as they were read from a data file. Examples: `E_NAME(0), E_NAME(log10(10)), E_NAME("somefield")`
• Field name	If no reserved function name is detected, and a word in the input does not match any of the above constants, it is interpreted as a field name. In a table function or selection context, a field name is interpreted to refer to the name of a table column. If a property name was detected, but the object the expression works on holds the same data already in a field, the data from the field is used. This means that it is for example to refer to a property via its name in the context of a table, if a table column with that data exists, even when the original ensemble which provided the data for the column does not exist any longer. Example: `col5`

The names of built-in functions are reserved and cannot be used as field or property names.

Numerical operators

Function operators use the standard precedence rules.

This is the set of built-in operators:

• -	Negation (unary operator)
• +	Addition of numerical data. If any of the arguments is a float, the result is a float, otherwise an integer.
• -	Subtraction of numerical data. If any of the arguments is a float, the result is a float, otherwise an integer.
• *	Multiplication of numerical data. If any of the arguments is a float, the result is a float, otherwise an integer.
• /	Division of numerical data. If any of the arguments is a float, the result is a float, otherwise an integer. Division by zero yields a `NULL` result.
• %	Modulo operator on integer data. Floating point values will be cast to integer. For *modulo* functionality with floating point conservation, use the `mod()` function.
• **	Exponentiation of float data. The result is always a floating point value.

Boolean operators

• *e1* <=> *e2*	Check whether the values of the expressions are equal. This comparison operator will return 1 if both values are **`NULL`**.
• *e1* = *e2*	Check for equality, with type coercion if necessary. **`NULL`** is by definition not equal to **`NULL`**.
• *e1* == *e2*	The same as the simple = operator.
• *e1* != *e2*	Check for inequality. **`NULL`** arguments will always result in a **`NULL`** result.
• *e1* <> *e2*	The same as the != operator.
• *e1* > *e2*	Check weather *e1* is larger than *e2*, with type coercion if necessary.
• *e1* < *e2*	Check weather *e1* is smaller than *e2*, with type coercion if necessary.
• *e1* >= *e2*	Check weather *e1* is larger than or equal to *e2*, with type coercion if necessary.
• *e1* <= *e2*	Check weather *e1* is smaller than or equal to e2, with type coercion if necessary.
• *x* between *l* and *h*	Check whether the value of *x* is between the low and high limit expressions *l* and *h*. The comparison values will be cast to the common type. The function may be used with string arguments. If *x* is within the range, the result is 1, otherwise 0.
• *x* in(*a1*,...)	If the value of *x* is equal one of the listed arguments, the result is 1, otherwise 0. If any of the arguments are **`NULL`**, a **`NULL`** *x* will also be found.
• not *e1*	Invert the boolean result of expression *e1*. Inverting **`NULL`** gives **`NULL`**.
• ! *e1*	This is an alias to the *not* comparison operator. However, ! may not be used as a replacement for *not* where *not* is a keyword, such is as a *not in* statement.
• *e1* and *e2*	Check whether both boolean input values are true. If any of the input arguments cannot be converted to an integer, or are **`NULL`**, the result is **`NULL`**.
• *e1* && *e2*	This is an alias to the *and* comparison operator. However, && may not be used instead of *and* in cases where *and* is a keyword, such as in *between* statements.
• *e1* or *e2*	Check whether any of the boolean input values are true. If any of the input arguments cannot be converted to an integer, or are **`NULL`**, the result is **`NULL`**.

• *e1* || *e2*	This is an alias to the *or* comparison operator.
• *e1* xor *e2*	Check whether exactly one of the boolean input values are true. If any of the input arguments cannot be converted to an integer, or are **`NULL`**, the result is **`NULL`**.
• *e1* ^ *e2*	This is an alias to the *xor* comparison operator.
• *x* is not null	Return 1 if expression *x* is not **`NULL`**, 0 otherwise.
• *x* is null	Return 1 if expression *x* is **`NULL`**, 0 otherwise.
• *x* not between *l* and *h*	This is the negation of the *between* range check.
• *x* not in(*a1*,...)	This is the negation of the **`in()`** function.
• *s* like *pat* [escape *c*]	Check whether string *s* matches the pattern *pat*. The pattern is mostly matched literally and is anchored to the left and right sides of the string. There are only three characters with special meaning in the pattern string: _ (underscore) matches one arbitrary character, and % (percent) matches any number of characters, including none. The special meaning of these characters in the pattern can be suppressed be prefixing it with the escape character, which is a backslash by default, but can be set by the optional phrase. The escape character may be escape by itself. The result of this function is a boolean match result value if none of the input parameters is **`NULL`**. The comparison is case-sensitive.
• *s* not like *pat* [escape *c*]	This is the negation of the *like* pattern match function above.
• *s* regexp *pat*	Perform an extended regular expression match of string *s* against regular expression pattern *pat*. The result is a boolean match value, or **`NULL`** if any of the arguments is **`NULL`**. The comparison is case-sensitive.
• *s* not regexp *pat*	This is the negation of above operator.
• *s* rlike *pat*	This is the same as the *regexp* operator.
• *s* not rlike *pat*	This is the negation of above operator.
• strcmp(s1,s2)	Compare strings s1 and s2. If s2 is lexicographically larger than s1, the result is -1. In the opposite case, the result is 1. If the strings are equal, the result is 0. The comparison is case-sensitive.

Bit operators

• ~	Bit-inversion (unary operator)
• |	Bit-or
• &	Bit-and

• ^	Bit-exclusive or
• <<	Leftshift
• >>	Rightshift (performed on unsigned argument)

All of them are identical to the C language definition, including precedence rules. They are not standard SQL.

Mathematical functions

This is the set of built-in numerical functions:

• abs(*x*)	Absolute value of float or integer *x*. The data type is preserved.
• acos(*x*)	Arc cosine of float *x*. If *x* is outside the range -1...1, **`NULL`** results.
• asin(*x*)	Arc sine of float *x*. If *x* is outside the range -1...1, **`NULL`** results.
• atan(*x*)	Arc tangent of float *x*.
• atan(*y*,*x*)	Arc tangent of *y/x*, using the signs of both parameters to determine the quadrant.
• atan2(*y*,*x*)	Arc tangent of *y/x*, using the signs of both parameters to determine the quadrant.
• bit_count(*x*)	Count the number of set bits in *x* after casting it to an integer.
• bitcount(*x*)	This is an alias to the **`bit_count()`** function.
• ceil(*x*)	Round float *x* up to the next integer. The result is an integer.
• ceiling(*x*)	Round float *x* up to the next integer. The result is an integer.
• clamp(*x*,*l*,*h*)	If *x* is less than *l*, it will be set to l. If it is larger than *h*, it will be set to *h*. The function preserves the data type of *x*.
• cos(*x*)	Cosine of float *x*.
• deg(*x*)	Convert float *x* from radians to degrees.
• degrees(*x*)	Convert float *x* from radians to degrees.
• double(*x*)	Force casting of argument *x* (can be int, float, string) into a float.
• exp(*x*)	Natural exponent of a float.
• float(*x*)	Force casting of argument *x* (can be int, float, string) into a float.
• floor(*x*)	Round float *x* downwards to the next integer. The result is an integer.
• int(*x*)	Force casting of argument *x* (can be int, float, string) into an integer.

• interval(*n*,*n1*,*n2*,...)	Compute the interval index of argument n. If *n* is smaller than *n1*, the result is 0. If it is between *n1* and *n2*, the result is 1, and so forth. If *n* is larger than any comparison value, the result is the number of comparison values. All arguments are cast to the type of *n*. This function can, in an extension of the SQL standard, used with all data types.
• irand(*x*)	Produce an integer random number between zero and **`x-1`**.
• irnd(*x*)	This is an alias for the function **`irand()`**.
• isnull(*x*)	If *x* is **`NULL`**, the result is integer 1, else 0.
• log(*x*)	Natural logarithm of float *x*.
• log10(*x*)	Decadic logarithm of float *x*.
• pow(*x*,*y*)	Raise *x* to the *y*th power. Both parameters are floats.
• power(*x*,*y*)	Raise *x* to the *y*th power. Both parameters are floats.
• rad(*x*)	Convert float *x* from degrees to radians.
• radians(*x*)	Convert float *x* from degrees to radians.
• range(*x*,*l*,*h*)	Check whether *x* is between (inclusive) the low and high limits *l* and *h*. The comparison values will be cast to the common type. If *x* is within the range, the result is 1, otherwise 0. This function may be used with string arguments.
• round(*x*)	Round float *x* to the next integer.
• round(*x*,*n*)	Round float *x* on the *n*th decimal place. Positive *n* indicates fractions after the decimal point, negative x rounds to next 10, 100, etc. A zero *n* is equivalent to the single-argument version of this function. If *n* is equal to or smaller than 0, the result is an integer, otherwise a float.
• rand()	Generate a floating point random number between 0 and 1. The random generator seed is not changed.
• rand(*x*)	Generate a floating point random number between 0 and 1. The random generator is seeded with integer argument *x*, and will, on consecutive calls, return the same sequence of pseudo-random numbers which is dependent on the seed argument *x*.
• rnd()	This is an alias for the function **`rand()`**.
• rnd(*x*)	This is an alias for the function **`rand(x)`**.
• sign(*x*)	The sign of float or integer *x*. The result is an integer -1, 0, or 1.
• sin(*x*)	Sinus of float *x*.
• sqrt(*x*)	Square root of float *x*. Roots of negative numbers will yield **`NULL`**.
• tan(*x*)	Tangent of float *x*.

• truncate(*x*,*n*)	Same as *round*(), except that truncation to the smaller absolute value is performed.

Date and time functions

• curdate()	Get the current time as a string in YYYY:mm:dd format.
• curtime()	Get the current time as a string in HH:MM:SS format.
• current_date	Get the current date as a string in YYYY-mm-dd format (ISO). This function does not use parentheses!
• current_time	Get the current time as a string in HH:MM:SS format. This function does not use parentheses!
• current_timestamp	Get the current date and time as a string in YYYY-mm-dd MM:HH:SS format (ISO). This function does not use parentheses!
• date_format(*f*,*x*)	Format the time specification *x* with a format string *f* interpreted by the **`strftime()`** C library function.
• dayname(*x*)	Get the English day name (Monday, Tuesday,...) from date specification *x*.
• dayofmonth(*x*)	Get the day of the month (starting with 1) from date specification *x*.
• dayofweek(*x*)	Get the day of the week from time specification *x*, using the ODBC encoding standard with 1=Sunday, 2=Monday, etc. Example: **`dayofweek('2003-1-1').`**
• dayofyear(*x*)	Get the day of the year (starting with 1) from time specification *x*.
• hour(*x*)	Get the hour from date specification *x* as an integer.
• minute(*x*)	Get the minute from data specification *x* as an integer.
• month(*x*)	Get the month number (1...12) from date specification *x*.
• monthname(*x*)	Get the English month name (January, February,...) from date specification *x*.
• now()	Get the current date and time as a string in YYYY-mm-dd HH:MM:SS format (ISO).
• quarter(*x*)	Get the quarter (1...4) from date specification *x*.
• second(*x*)	Get the second from date specification *x* as an integer.
• sysdate()	Get the current date and time as a string in YYYY-mm-dd HH:MM:SS format (ISO).
• time()	Get the current time as number of seconds since Jan 1st, 1970 as an integer value.

• time(*x*)	Decode time specification string *x* and return the value as seconds since Jan 1st, 1970. The types of dates which can be parsed depend on whether the toolkit was compiled with Tcl support or not. The toolkit directly parses a number of standard formats, such as ISO dates and times, but not any locale-dependent formats such as British/US dates. If the toolkit is compiled with Tcl as scripting language, the Tcl time/date parser will be used in addition to the built-in parser. Example: `time('2003-1-15')` as ISO data will be understood with or without Tcl support.
• time_format(*f*,*x*)	Format the time specification *x* with a format string *f* interpreted by the `strftime()` C library function.
• unix_timestamp()	Get the current time as number of seconds since Jan 1st, 1970 as an integer value.
• unix_timestamp(*x*)	This function name is an alias to the *time(x)* function.
• week(*x*)	Get the week number (0...53) from date specification *x*. The week is assumed to begin with Sunday.
• week(*x*,*mode*)	Get the week number from date specification *x*. Mode can be one of 0: week starts on Sunday, week range is 0...53; 1: week starts on Monday, week range is 0...53; 2: week starts on Sunday, week range is 1...54; 3: week starts on Monday, week range is 1...54.
• weekday(*x*)	Get the day of the week index from date specification *x*. Here Sunday=0, Monday=1, etc.
• year(*x*)	Get the year as integer (including centuries) from date specification *x*.
• yearweek(*x*)	Get year and week as a six-digit integer in format YYYYWW from date specification *x*. The week begins with Sunday.
• yearweek(*x*,*start*)	Get year and week as a six-digit integer in format YYYYWW from date specification *x*. For *start* value 0, the week begins on Sunday, for value 1 on Monday.

String functions

• ascii(*s*)	Get the ASCII/ISO code of first character of string *s*. If the string is empty, the result is 0.
• bit_length(*s*)	Get the length of string *s* in bits. In this implementation, this is always the number of characters in *s* multiplied by 8.
• char(*c1*,*c2*,...)	Interpret the arguments as ASCII/ISO character codes and construct a string which is the concatenation of all characters.
• char_length(*s*)	Get length of string *s*.
• character_length(*s*)	Get length of string *s*.

- color(*v*,*vmin*,*vmax*,*ncolorshades*,*color1*,*color2*,...)
  This function will compute a color value and return it as an X11 color specification in the format *#rrggbb*. The relative position if floating-point input value *v* between the minimum and maximum values *vmin* and *vmax* is computed. If *v* is outside the range, it is set to the closest boundary value. The relative position is then assigned to the corresponding color spaces from *color1*...*color2*, *color2*...*color3* (if a third color is specified) and so on. All color spaces have the same width, so if there are two color spaces, the first color space is used if v_{rel} is between 0 and 0.5, and the second one if it is larger. A new relative position is then computed within the color space, so if v_{rel} is 0.25 and there are two color spaces, it is placed halfway into the first color space. Every color space is partitioned into *ncolorshades* different shades by linear interpolation of the RGB values of the corner colours. The corner colours may be provided as X11 color database names, or in X11 RGB notation. The final result of the function is the shade in the appropriate color space v_{rel} is positioned on. For a simple grayshade interpolation, only *white* and *black* need to be passed as corner color pair. For a rainbow scheme, use *red*, *green* and *blue* as a color triple. This function is not a standard SQL function.
- concat(*s1*,*s2*,...) Concatenate the argument strings. If any argument is **`NULL`**, the result will be **`NULL`**.
- concat_ws(*sep*,*s1*,*s2*,...) Concatenate the argument strings, and insert the separator string *sep* between them. There will be no separator before the first component string, or after the last. Empty concatenation strings and **`NULL`** strings will be skipped. If the separator is **`NULL`**, the result is **`NULL`**. If the separator is an empty string, the result is identical to the simple **`concat()`** function.
- conv(*s*,*fbase*,*tbase*) Convert the string *s*, which is interpreted as being an unsigned number in base *fbase* (an integer between 2 and 36). *s* may also directly be provided as a in integer parameter, in which case the decoder base is ignored. If the first input parameter is a string, it is decoded with the specified base. A new string with the value re-encoded in base *tbase* (an integer between 2 and 36) is generated. This function does not provide the full capabilities of the SQL function, because it currently does not handle signed numbers. If the target base is 16, and the input is a string, the string will be (in a diversion from the SQL standard) encoded as a hex-encoded string.

- export_set(*bits*,*on*,*off*[,*sep*][,*nbits*])
  Construct a string from a bit-encoded integer value *bits*. The encoding starts with the LSB bits, moving upwards. For every set bit, the *on* string value is concatenated to the output string and for every unset bit the *off* string value. Bit positions in the output string are separated by a separator string, which is a comma character by default and may be changed by providing the first optional parameter. By default, all 32 bit positions of the standard toolkit integers are processed, but this number may be adjusted by the last optional parameter. All element and separator strings may be set to an empty string in order to omit them.
- find_in_set(*s*,*set*) Try to find string *s* in a string-encoded set *set*. Set elements are separated by a comma. Search string *s* must not contain commas. The function returns 0 if the string cannot be found, or the set is empty, otherwise the set element position starting with 1. If any of the arguments is **`NULL`**, the result is **`NULL`**.
- insert(*s*,*p*,*l*,*snew*) Remove *l* characters from string *s*, starting at position *p* (which begins at 1), and replace the removed sequence with string *snew*. If the length *l* is zero, the new string is simply inserted at the requested position. A minimal position value of 1 is silently enforced. Inserting beyond the length of the string is the same as a simple concatenation.
- instr(*s*,*substr*) This is the same function as **`locate()`**, but the arguments are swapped. We love SQL.
- lcase(*s*) Concert string *s* to lower case.
- load_file(*f*) Load file *f* and return its content as a string.
- left(*s*,*l*) Return the leftmost *l* characters from string *s*. If *s* is shorter than *l*, *s* will be passed unchanged.
- length(*s*) Get length of string *s*.
- locate(*substr*,*s*) Locate the position (starting with 1) of the first occurrence of substring *substr* in string *s*. If the substring is not found, 0 is returned.
- locate(*substr*,*s*,*p*) Locate the position (starting with 1) of the first occurrence of substring *substr* in string *s*, beginning the search at position *p* (also starting with 1). If the substring is not found, 0 is returned.
- lower(*s*) This is an alias for the **`lcase()`** function.

• lpad(*s*,*l*,*pad*)	Left-pad string *s* by repeating string *pad* until a length of *l* is reached. The pad string may be longer than one character, but then only a part of the pad string may be used. The the string is already longer than *l*, it will be truncated. If the pad string is empty or **`NULL`**, a space character will be used for padding.
• ltrim(*s*)	Remove leading whitespace from string *s*.
• make_set(*bits*,*s1*,*s2*...)	Construct a string-encoded set by concatenating those element strings which correspond to set bits in the bits argument with a comma. The set element strings should not contain commas themselves. If bit 1 is set, string *s1* is used, bit 2 decides whether s2 is included, and so forth.
• mid(*s*,*p*,*l*)	Return *l* characters of string *s*, starting with position *p* (which begins with 1). If the remaining string after position *p* is shorter than *l*, the rest of the string will be returned. If *p* is larger than the string length, an empty string is produced. If *p* is smaller than one, it is implicitly set to 1.
• octet_length(*s*)	Get length of string *s*.
• ord(*s*)	Get ASCII/ISO code of first character of string *s*. If the string is empty, the result is `0`.
• position(*substr* in *s*)	Locate the position (starting with 1) of the first occurrence of substring *substr* in string *s*. If the substring is not found, 0 is returned. Note that the parameters are separated by the keyword *in*, not a comma!
• regsub(*s*,*pat*,*rpl*[,*all*])	Perform a regular expression substitution on input string *s*. *pat* is an extended regular expression which is matched in case-sensitive fashion. When a match is found, the matched part of the input string is replaced by the *rpl* pattern. Within *rpl*, the usual regular expression replacement operators & (full matched string section) and \1...\9 (matched bracketed sub-expressions of the pattern) are recognized. These replacement operators may be escaped by a backslash character in order to prevent their interpretation. By default, only the first match in the input string is substituted. The process may be changed to global substitution by passing a *true* boolean *all* parameter as optional argument. The return value is the substituted string. If no match occurred, the original string is copied unchanged. This function is not part of the normal SQL function set.
• repeat(*s*,*n*)	Concatenate string *s* *n* times and return the result. If *n* is equal to or less than zero, and empty string is produced. When *s* or *n* are **`NULL`**, the result is also **`NULL`**.
• replace(*s*,*f*,*t*)	Replace all occurrences of substring *f* in string *s* by string *t*.

• reverse(*s*)	Invert the sequence of characters in string *s*.
• right(*s*,*l*)	Return the rightmost *l* characters from string *s*. If *s* is shorter than *l*, *s* will be passed unchanged.
• rpad(*s*,*l*,*pad*)	Right-pad string *s* by repeating string *pad* until a length of *l* is reached. The pad string may be longer than one character, but then only a part of the pad string may be used. The the string is already longer than *l*, it will be truncated. If the pad string is empty or **`NULL`**, a space character will be used for padding.
• rtrim(*s*)	Remove trailing whitespace from string *s*.
• soundex(*s*)	Generate *soundex* string from input string *s*. Soundex strings allow phonetic comparison of (preferably English) words and phrases.
• string(*x*)	Cast argument *x* to a string. If *x* is already a string, the function does nothing.
• substring(*s*,*p*)	Get all characters of string *s* after position *p* (beginning with 1). If *p* is smaller than 1, it is silently set to 1. If *p* is beyond the length of the string, an empty string is produced.
• substring(*s* from *p*)	The same as above function, just using the keyword from instead of a comma as separator.
• substring(*s*,*p*,*l*)	This is the same as the **`mid()`** function.
• substring(*s* from *p* for *l*)	This is the same as the **`mid()`** function, but the arguments are separated by the keywords *from* and *for*.
• substring_index(*s*,*delim*,*n*)	Return the substring of string *s* before the delimiter character delim is found **`abs(n)`** times. If *n* is positive, the string is scanned from left to right, otherwise in reverse direction. If *n* is 0, an empty string is produced. If the delimiter is **`NULL`** or an empty string, the result is **`NULL`**. Only the first character from the delimiter string is used.
• trim(*s*)	Remove leading and trailing whitespace from string *s*.
• trim(*r* from *s*)	Remove leading and trailing instances of string *r* from string *s*.
• trim(both from *s*)	Remove leading and trailing whitespace from string *s*.
• trim(leading from *s*)	Remove leading whitespace from string *s*.
• trim(trailing from *s*)	Remove trailing whitespace from string *s*.
• trim(both *r* from *s*)	Remove leading and trailing instances of string *r* from string *s*.
• trim(leading *r* from *s*)	Remove leading instances of string *r* from string *s*.
• trim(trailing *r* from *s*)	Remove trailing instances of string *r* from string *s*.
• ucase(*s*)	Convert string *s* to upper case.

- upper(*s*) This is an alias to the **`ucase()`** function.

Context functions

- file() Get the name of the file which is currently processed in the expression context. If no file is process, **`NULL`** is returned.
- record() Get the current record (file/database context) or row (table context) number. Numbering begins with 1.
- row() This is an alias to the **`record()`** function, which is more readable in the context of table operations.
- session_user() Get name of user.
- system_user() Get name of user.
- user() Get name of user.

Argument selection and flow control functions

- *a1* ? *a2* : *a3* Return the second argument if *a1* (cast to an integer) is not 0, otherwise return *a3*. If a1 is **`NULL`**, the result is also **`NULL`**, regardless of the values of *a2* and *a3*. This is an SQL extension which was modelled after the C language construct.
- case *x* when *c1* then *r1* [when *c2* then *r2* ...] [else *r99*] end
  Compare expression value *x* with the comparison expression values *c1*...*cn*. If any of them is equal to *x*, the corresponding result value *r* is returned. If none is equal, the return value of the optional else part is returned. If there is no else part, the result is **`NULL`**. The comparison values are cast to the type of *x*.
- case when *e1* then *r1* [when *e2* then *r2*...] [else *r99*] end
  This is a variant of the *case* statement. Here, individual expressions *e1*...*en* are evaluated and interpreted as boolean values. If any of them is true, the corresponding return value is extracted. If none of the expressions yields a true result, the optional else part is returned, or **`NULL`** if no else part was provided.
- coalesce(*a1*,...) Return the first argument which is not **`NULL`** with its original data type. If none of the arguments meets this criterion, **`NULL`** is returned.
- elt(*n*,*s1*,...) If *n* is 1, the first string argument is returned, the second string if *n* is 2, and so on. If *n* is outside the range of supplied strings, the result is **`NULL`**.
- field(*s*,*s1*,...) Return the index (beginning with 1) of the string in the string list beginning with *s1* which is identical to string *s*. If the first string is not found anywhere in the list, the result is 0.

• fieldref(*x*)	Force interpretation of string or expression *x* as a field/column reference. If the reference cannot be resolved, the result is **`NULL`**, otherwise the field value in the expression context. This is an SQL extension.
• if (*a1*,*a2*,*a3*)	If the value of expression *a1* is not an integer zero (after casting, if necessary) and not **`NULL`**, the result of expression *a2* is passed on, else the result of expression *a3*.
• ifnull(*a1*,*a2*,...)	This function passes the first non-null argument expression value on. If there are no non-**`NULL`** arguments, the result is **`NULL`**. Standard SQL provides this function only with exactly two arguments.
• largest(*a1*,...)	This is an alias for function **`greatest()`**.
• least(*a1*,...)	Select the smallest argument. The arguments can be either integers, floats, or strings. If all arguments are integers, the result is an integer. If any float is used, and no strings are involved, the result is a float and all arguments are compared as floats. If any argument is a string, all arguments are converted to strings and case-sensitive string comparison is used. Note that the **`min()`** function an as aggregate function is in a totally different class and no substitute for this function.
• greatest(a1,...)	Same as *least()*, except that the greatest argument is selected instead of the least.
• nullif(*a1*,*a2*)	If the result of expression *a1* is the same as *a2*, the return value is **`NULL`**. Otherwise, the result of expression *a1* is passed on. The comparison values are cast for comparison if necessary.
• property(*x*)	Force interpretation of string or expression *x* as a property reference. If the property name cannot be resolved, the result is **`NULL`**, otherwise the value of the property data in the expression context. This is an SQL extension.
• propref(*x*)	This is another name for the **`property()`** function.
• smallest(*a1*,...)	This is an alias for function **`least()`**.

Aggregate functions

• average(*field*)	This is an alias for the **`avg()`** function.
• avg(*field*)	Get the average of the values of all non-**`NULL`** data items in the selected field. If no such item can be found, the result is **`NULL`**, otherwise a float value. This function can only be used on numerical fields/columns or strings which can be cast to numbers.
• count(*)	Count the number of records or rows in the data object. No object values are used for the comparison.
• count(*field*)	Count the number of non-**`NULL`** values for the specified field in the data object.

• max(*field*)	Get the maximum value among all non-**`NULL`** data items in the selected field/column. If no such item can be found, the result is **`NULL`**, otherwise the minimum value in its original data type.
• min(*field*)	Get the minimum value among all non-**`NULL`** data items in the selected field/column. If no such item can be found, the result is **`NULL`**, otherwise the maximum value in its original data type.
• prod(*field*)	This is an alias for the **`product()`** function.
• product(*field*)	Get the product of the values of all non-**`NULL`** data items in the selected field. If no such item can be found, the result is **`NULL`**, otherwise a float value. This function can only be used on numerical fields/columns or strings which can be cast to numbers.
• squaredsum(*field*)	This is an alias for the **`sqsum()`** function.
• sqsum(*field*)	Get the sum of the squared values of all non-**`NULL`** data items in the selected field. If no such item can be found, the result is **`NULL`**, otherwise a float value. This function can only be used on numerical fields/columns or strings which can be cast to numbers.
• std(*field*)	This is an alias for the **`stddev()`** function.
• stddev(*field*)	Get the sum of the squared values of all non-**`NULL`** data items in the selected field. If no two such items can be found, the result is **`NULL`**, otherwise a float value. This function can only be used on numerical fields/columns or strings which can be cast to numbers.
• sum(*field*)	Sum up the values of all non-**`NULL`** data items in the selected field. If no such item can be found, the result is **`NULL`**, otherwise a float value. This function can only be used on numerical fields/columns or strings which can be cast to numbers.

CACTVS Python Scripting Overview

The Cactvs toolkit has both a Tcl and a Python scripting interface. Both interpreters are active at the same time. It is possible to mix script languages, for example by using Tcl-scripted property computations with a Python main script, or vice versa.

The capabilities of the Python interface are almost identical to the Tcl version. The only exception are thread-related commands. Python still does not support effective multi-threading and has severe limitations on this front, including lack of support fore the use of subinterpreters together with threads.

The Python commands have generally the same syntax has their Tcl equivalents, with the same order of arguments. In addition, all functions have parameter names, so that it is possible to omit unused arguments without entering explicit empty place-holders, as in the Tcl interface.

The function or method arguments of the toolkit interface are interpreted in a more lenient fashion than the standard Python behaviour. As in Tcl, automatic argument type conversion is attempted. Argument conversions from string into more complex types such as lists or dictionaries parse the simpler Tcl syntax for the string form of these types, not the corresponding Python string form.

Example: an expected list argument can be passed in as a true Python list, or as string encoded in the form "**`a b c`**", which is the Tcl form, but not as "**`['a','b','c']`**" which would be the native Python string representation. Commands which expect Cactvs object references accept both a native Python wrapper object, or the string form of the object handle. In any case, this extended argument parsing only applies to the toolkit-specific commands, not the generic Python commands.

Major Object Classes

The interface to the major chemistry objects is implemented as classes. The name of the class is the same as the object name or associated command name in the Tcl interface, except that the first letter is uppercase. All toolkit functions are located in the **`pycactvs`** namespace.

Example: **`pycactvs.Ens, pycactvs.Molfile, pycactvs.Dataset`**

The bundled toolkit Python interpreter already imports everything from the **`pycactvs`** namespace, so instead of **`pycactvs.Ens`** the class can be accessed in a simpler fashion as **`Ens`**. If the toolkit is used as a loadable Python module, the import is not automatic. In that context, either the fully qualified class name is used, or an explicit import like

```
from pycactvs import *
```

needs to be run.

All class-specific Tcl subcommands are accessed as object or class methods in Python. Class methods start with an uppercase character, while object methods use lowercase. Except for this distinction, the method names are the same as the respective Tcl subcommands. Additionally, a standard Python object constructor is available.

Constructors and Reference Objects

In the Python interface, objects are usually created by constructors, not by **`create`** or **`open`** object subcommands. The constructor arguments are the same as the Tcl create/open subcommands.

Example:

```
e=Ens("cccccn1")
m=Molfile("z.smi","w")
```

The return value of the constructor is a reference object. Its print format is the same as a Tcl object handle, but internally it is a different type of object.

It is also possible to create a reference object for an existing chemistry object via the **`Ref`** class method, which is implemented for all chemistry objects. This method takes a string form of the object handle, which could for example have been obtained from a Tcl function.

Example:

```
e=Ens.Ref("ens0")
```

For maximum compatibility with the Tcl syntax, the major chemistry object classes also have class-level factory methods under the name **`Create`** (and alias **`Open`** for **`molfile`** objects).

Example:

e=Ens.Create("c1ncccc1","smarts")

The result from the factory function is exactly the same as from the equivalent constructor.

Object Deletion

Deleting a reference object does not automatically delete the chemistry object.

Example:

```
e=Ens("c1ncccc1")
del e
```

The ensemble persists even in the absence of a reference object. The **`del`** command removes the reference object, but not the underlying toolkit ensemble object. Explicit deletion (or some other mechanism like with-clauses, see later) is required:

```
e=Ens("c1ncccc1")
e.delete()
del e
```

On the other hand, chemistry objects cannot be fully deleted as long as there are reference objects to them. Reference objects contain an object pointer which must not become invalid. In above example, using the **`e`** variable after the deletion will raise an error, but not crash the application. If reference objects to invalid chemistry objects are not explicitly deleted, a minimal chemistry object frame must be retained. This can lead to accidental accumulation of object zombies, which are not resolved until all references have gone out of scope and are garbage-collected. In extreme cases, this can lead to memory problems.

In case an object reference is not available, or larger object collections need to be disposed of, all objects also support deletion via a class function:

```
Ens.Delete(e,"ens1")
```

Above command deletes the ensemble referenced by variable **`e`**, as well and the ensemble which has handle *ens1*.

Minor Object References

Atoms, bonds, rings, or network connections and vertices - all examples of minor chemistry objects which exist only in the context of a major chemistry object such as an ensemble or network - also use reference objects as access mechanism. Chemistry object methods which return atoms, bonds, etc. generate minor object reference objects, not simple numeric object labels. Example:

```
e=Ens("CCC")
alist=e.atoms(filters="carbon")
print(alist)
print(alist[0].ens())
```

The atom list prints as "[1,2,3]". However, the objects are not integers, but atom references, for which the default print format is their label value. It is still possible to obtain their referring ensemble object, which is not possible for the equivalent Tcl result list. Like major object references, minor object references also block full deletion of the referred major chemistry object until all reference objects to that chemistry object have been collected. Minor object references can be converted to a true integer label with the **int()** function (see paragraph on the numerics protocol).

It is possible to obtain minor object references via a numerical label (or other minor object identifier, see Tcl documentation of **atom atom**, **bond bond** etc.) and the major object reference:

```
e=Ens("CCC")
a=Atom.Ref(e,1)
```

or by the minor object resolution commands of the associated major object:

```
a=e.atom(1)
a=e.atom("#2")
```

These commands get a reference to the atom with label one, and the third atom (from zero-based index 2) in the atom list, regardless of its label.

Finally, atom references can be obtained from ensemble references by using a numerical index, but no other atom identifiers. These are zero-based atom list indices, not labels. The same feature is supported for vertices of networks. Atom slices can be obtained in the same way, though this is not that useful in practical applications. It is not possible to get references to other minor objects (such as bonds, rings, or network connections) via this scheme.

```
a=e[1]
a=e[1:3]
```

Minor chemistry references store a state stamp of the major object they refer to. As long as the major object state does not change (such as by deleting an atom), they can directly access their minor object via a pointer, without resolving the label. In case of a state change, the label is resolved again. That operation can fail, for example by the deletion of an atom between the time the reference was established, and when it is used again.

Minor object references can still be used when there is a mismatch between the referred major object, and another major object where it is used as an object identifier. An example would be the an atom reference which represents a structure match on one structure, whereupon it is used to access an atom with the same label in a template structure. In such contexts, the corresponding atom in the second structure is automatically and implicitly looked up via the label component. The reference object remains bound to its original structure.

Controlling Major Object Lifetimes by means of *with* Statements

Ensembles, reactions, datasets, tables, networks, molfiles, database connectors, the bitvector objectt as well as **SOAP**/**XML** and **JSON** parser objects implicitly and automatically support the **`__enter__/__exit__`** Python object protocol. This means, they can be used in the context of with statements.

An object defined in the context of a *with* statement is automatically deleted as soon as the statemens goes out of scope. This is even true if the statements under the *with* clause are not fully executed, for example because an exception is thrown.

Example:

```
with Ens("c1ncccc1") as e:
    print(e.E_WEIGHT)
```

The ensemble is deleted immediately after the print statement. In case there are other references to the object outside the *with* context, the same caveats as for explicit object deletion apply.

Property and Filter References

While properties and filters are not chemistry objects, they use a reference mechanism which is very similar to that of major objects. Interface commands that return properties or filters return reference objects, not strings. In interface commands, it is generally possible to use either strings of references in arguments where filters or properties are expected.

```
p=Prop.Ref(“E_WEIGHT”)
print(p.datatype)
e=Ens(“c1ncccc1”)
print(e.get(“E_WEIGHT”))
print(e.get(p))
f=Filter.Ref(“carbon”)
print(f.property)
print(e.atoms(“carbon”))
print(e.atoms(f))
```

Since in many cases there are not many commands to be executed on a filter or property, class methods which perform routine operations without a reference object are also implemented:

```
print(Prop.Get(“E_WEIGHT”,”unit”))
Prop.Set(“E_SMILES”,”parameters”,”usearo 0”)
```

References can also be obtained by means of the definition file input methods, which are implemented as class methods:

```
p=Prop.Read(“myprop.xpd”)[1]
f=Filter.Readblob(somexmldata)
```

These file input commands can read files which contain more than one definition, so the return value is a tuple where the first item is the record count, and the second value the reference to the first decoded record. This is the reason for the `[1]` index of the property reader.

Property Data Retrieval

All chemistry objects provide the standard set of data retrieval commands, such as **get**, **new**, **dget**, **nget**, **jget**, etc. They are used in the same fashion as the **TCL** equivalents:

```
e=Ens("c1ncccc1")
print(e.get("E_WEIGHT"))
p=Prop.Ref("E_WEIGHT")
print(e.get(p))
a=e.atom(1)
print(a.get("A_ELEMENT"))
```

In addition, the plain form of the get command is also accessible via a direct attribute access:

```
print(e.E_WEIGHT)
```

This can also be written like access to a dictionary:

```
print(e["E_WEIGHT"])
```

All standard toolkit property names have been verified to be syntactically compatible with the direct notation, and changed in case they did not pass the required name syntax check. A couple of deprecated property name forms remain valid as default alias definitions. These can be used as string property names in **get()** or dictionary-style access, but not as direct access attributes.

The use of additional parameters or filters for data retrieval requires the use of full **get** (and related) statements:

```
print(e.get("A_ELEMENT","chargedatom"))
print(e.get("E_SMILES",None,"usearo 0 unique 1")
print(e.get("E_SMILES",parameters={"usearo":False,"unique":True})
```

The last line is the same command as the one in the preceding line (which follows closely the **TCL** standard syntax), just written in a more Pythonic form.

Access to property data fields, special encodings, or other indexing features likewise requires the use of a full retrieval command, with a few exceptions.

```
print(e.get("E_GIF(dataurl)"))
```

Above statement, which returns the image as a HTML data URL instead of the standard property data type (external disk file or blob), cannot be written

For vector types, indexing and slicing can be used in a very similar fashion. However, the internal toolkit indexing include the upper index in a slice, while the Python version does not.

```
print(e.get("E_NAMSET(0)"))
print(e.E_NAMSET[0])
```

The two commands return the same result. However,

```
print(e.get("E_NAMSET(0:2)"))
```

returns names with index 0 up to and including 2 (3 elements), while the Python construct

```
print(e.NAMESET[0:2])
```

returns only two names, for indices 0 and 1. The same rules apply to slices of Latin1 and Unicode strings. Note that indexing of strings (but not slicing) in the toolkit accesses words in the strings (the separator characters are configurable for each property), which is different from simple character access when using Python element access syntax.

Property data of *compound* type is a special case. Results are returned as a named tuple, which is automatically constructed from the property definition. Field in the tuple can be directly accessed both via index and field name.

```
print(e.E_BRUNS_WATSON_DEMERIT)
```

The Bruns-Watson demerit data consists of two fields: *demerit*, which contains the overall demerit score, and *features*, which is a dictionary of the features which were used in scoring. The keys are the feature names, the values to individual demerit values. These can all be accessed directly:

```
print(e.get(“E_BRUNS_WATSON_DEMERIT(demerit)”))
print(e.E_BRUNS_WATSON_DEMERIT[0])
print(e.E_BRUNS_WATSON_DEMERIT.demerit)
```

All three commands return the same result. The property data access can be nested.

```
print(e.E_BRUNS_WATSON_DEMERIT.features[“not_enough_atoms”])
```

Property Data Modification

All access methods to the *complete* property data content of an object can also be used for setting property data.

```
e=Ens(“c1ncccc1”)
e.set(“E_NAME”,”pyridine”)
e.E_NAME = “pyridine”
e[“E_NAME”] = “pyridine”
```

However, when manipulating individual fields or elements of a property record, only indexing the property name in a **`set()`** command has the desired effect.

```
e.E_NAMESET = (“name1”,”name2”,”name3”)
e.set(“E_NAMESET(1)”,”name2_new”)
e.E_NAMESET[2] = “name3_new”
```

The last command line fails, while the first two succeed. The reason is that once data has been exported from the library to Python, after a retrieval command, but before any Python-side indexing and element access operations, it is stored in these objects and they are decoupled from the original property data in the library objects. So either the command only manipulates the Python-side language objects, or the operation even fails because, for example, tuples are immutable.

Retrieval and Modification of Object Attributes

Object attributes can be queried and modified both by explicit **`get`**/**`set`** functions and by direct access. This works in thed same fashion both for chemistry objects (ensembles, datasets, etc.) and auxiliary objects (properties, filters, database connectors, etc.).

Example:

```
print(Prop.Get("E_NAME","parameters"))
p=Prop.Ref("E_NAME")
print(p.get("parameters"))
print(p.parameters)
Prop.Set("E_NAME","parameters",{"pubchemlookup":True,"lmchlookup":False})
p.set("E_NAME","parameters",{"pubchemlookup":True,"lmchlookup":False})
p.parameters={"pubchemlookup":True,"lmchlookup":False,"useformula":True}
```

All standard object attributes have been checked for syntactic compatibility. In same cases, the default name was changed during the review to support this. Older attribute names can still be used, but only within spelled-out **`get`**/**`set`** commands, where the syntax is more flexible.

Class Methods as Replacement for Single-Shot Commands

Some of the Tcl interface chemistry object manipulation commands support single-shot commands. In these, instead of the object handle identifying the object the command should work on, the command also accepts simple **`constructor/create()`** arguments. Such commands automatically destroy the object they temporarily create for the execution of the command once it has completed, even in the contexts of errors.

```
puts [ens get c1ncccc1 E_WEIGHT]
puts [molfile count somefile.smi]
puts [dataset [list $eh1 $eh2 $eh3] get D_CHEMDRAW_PAGE {} {filename mypage.cdx}]
```

These are all examples of single-shot commands. Because in Tcl the object argument is decoded anew for every command, and this happens for every subcommand at the beginning of the command execution, any single ens/molfile/dataset commands supports single-shot operation.

In the Python interface, commands are by default implemented as object methods. The objects which execute the methods have already been fully decoded (see the paragraph on constructors and factory class methods). This means, they cannot provide single-shot methods.

These can however be implemented as class methods - at the cost of handling every such command by a separate function code. For this reason, only the most commonly used single-shot command variants have been implemented. Their name is the same as the object method name, except that they begin with an uppercase letter. They accept a simple single constructor argument as the first parameter, and all subsequent parameters are the same as for the object method. The Tcl examples from above can be coded in Python as

```
print(Ens.Get(“c1ncccc1”,”E_WEIGHT”))
print(Molfile.Count(“somefile.smi”))
print(Dataset.Get((e1,e2,e3),”D_CHEMDRAW_PAGE”,None,{“filename”:”mypage.cdx”})
```

The following single-shot class methods are currently implemented for chemistry objects. This list excludes the **`Ref(), Create() and Delete()`** class methods supported for all objects, as well as **`Unpack(), Exists(), Defined()`** and **`List()`**. For database connectors and molfiles, the **`Close()`** method is an alias for **`Delete()`**. These methods are not single-shot commands.

```
Dataset.Get() and variants Dget(), Jget(), Jnew(), New(), Nget(), Nnew(), Show()
Dataset.Scan()
Dataset.Transform()
Ens.Get() and variants Dget(), Jget(), Jnew(), New(), Nget(), Nnew(), Show()
Majorobj.Ldup()
Majorobj.Lhdup()
Molfile.Add()
Molfile.Count()
Molfile.Get() and variants Dget(), Jget(), Jnew(), New(), Nget(), Nnew(), Show()
Molfile.Fullscan()
Molfile.Hloop()
Molfile.Hread()
Molfile.Loop()
Molfile.Max()
Molfile.Min()
Molfile.Peek()
Molfile.Read()
Molfile.String(), and alias Molfile.Blob()
Molfile.Scan()
Molfile.Truncate()
Molfile.Write()
Reaction.Get() and variants Dget(), Jget(), Jnew(), New(), Nget(), Nnew(), Show()
```

Some objects have input class methods, which are also different from single-shot methods. These are essentially constructors and comparable to the class **`Create()`** object factory methods.

```
Dataset.Unpack()
Ens.Unpack()
Network.Read()
Network.Unpack()
Reaction.Unpack()
Table.Read()
Table.Readblob()
Table.Unpack()
```

Another special case are class methods which modify a collection of minor objects in a single command:

```
Atom.Delete()
Atom.Dup()
Atom.Hdup()
Atom.Xdelete()
Bond.Delete()
Bond.Xdelete()
```

Pickling and Unpickling

The ensemble, reaction, dataset, table, network and bitvector objects support the standard Python pickle/unpickle protocol. The data stored is the same as the **`pack()`** method with standard arguments.

Example:

```
import pickle
with open('pickledstructures','wb') as p:
        with Ens('c1ncccc1') as e:
                pickle.dump(e,p)
p=open('pickledstructures','rb');
e=pickle.load(p);
print(e.E_SMILES)
```

Object Iterators

The standard Tcl object loop commands (**`molfile loop, molfile hloop, dataset loop, ens loop, reaction loop, table listloop, table dictloop, table rownamelistloop, table rownamedictloop`**) map to Python syntax only in a very awkward fashion. There is a direct implementation which accepts the loop body as either as multi-line string, or a function reference. This is the string code style:

```
d=Dataset('c1ncccc1','c1ncncc1')
d.loop('''
print(e.E_WEIGHT)
''',variable='e')
```

That definitely does not look like Python, and in the string loop body indentation must begin with 0, adding another layer of ugliness.

The function style looks much cleaner, but it breaks the program code flow:

```
def body(e):
        print(e.E_WEIGHT)

d=Dataset('c1ncccc1','c1ncncc1')
d.loop(body)
```

A much more Pythonic style it the use of iterators. Molfiles, datasets, ensembles and tables can provide standard Python iterators. These objects acts as their own iterators in the *self* iterator style. The dataset loop then looks like:

```
d=Dataset('c1ncccc1','c1ncncc1')
for e in d:
        print(e.E_WEIGHT)
```

One important caveat with *self*-style iterators is that they cannot be nested. At any time, there exists only a single iterator per object. Using them simultaneously in multiple loops will have unexpected consequences.

Tables and molfiles offer multiple loop types, but there is only a single iterator. For molfiles, there is no specific iterator type control. The different effects of the **`loop`** and **`hloop`** loop styles can be mirrored in the iterator by setting the *hydrogens* read attribute appropriately. For tables, the default iterator style is *list*, i.e. the row content is presented as a tuple. Other table iterator styles can be configured by setting the *iteratorstyle* attribute.

```
tp.iteratorstyle = 'dict'
```

Mapping Protocol Object Functions

All of the chemistry objects support parts of the Python mapping protocol.

Reaction support the len() function, which tells the number of ensembles in the reaction. In addition, it supports getting and setting object properties and attributes via this protocol, which syntactically looks like accessing a dictionary:

```
x=Reaction("C=C>>CC")
x.X_NAME="hydrogenation"
print(x.X_NAME)
print(x["X_NAME"])
print(len(x))
```

There is no difference between the two alternative methods. The dictionary style has the advantage that property and attribute names may be used which cannot be used in direct access due to language syntax limitations.

Other major chemistry objects use the same data access methods. The only difference is the meaning of the **len**() function. For ensembles, it returns the number of atoms, for datasets, the number of objects in the dataset, for molfiles the number of records, for tables the number of rows, and for networks the number of vertices.

All minor chemistry objects also support the mapping protocol. The length function returns the number of other minor objects this object links (e.g. the number of bonds in an atom, the number of atoms in a bond, or the number of atoms in a ring or molecule). Property retrieval and setting via dictionary-style access is the same as for major objects.

Property and filter references support attribute manipulation, but no **len**() function.

References to structure file I/O modules, table I/O modules, network I/O modules, datatype handler modules, and database connector modules all support attribute reading in the mapping fashion. Only the structure file and table I/O modules allow attribute setting.

Bitvectors are another non-chemical object type which support the mapping protocol. All three functions (length, indexed or sliced value retrieval, indexed value setting) are implemented.

Sequence Protocol Object Functions

Datasets and reactions support parts of the Python sequence protocol.

The **`len()`** associated with the protocol function returns the same values as for the mapping protocol.

Reactions and datasets additionally support the **`in`** test to check whether an object is contained in the sequence, the access of elements via index or slice, and the += in-place concatenation method. The latter is the same function as the number protocol function.

```
d=Dataset(“C=C”,”CC”)
eitem=d[0]
print(eitem in d)
etuple=d[0:2]
```

The code example prints **`True`**.

Number Protocol Object Functions

Several of the chemistry objects have overloaded numerical operators.

Minor Objects only support the **`int()`** and **`float()`** functions. The result is the object label transformed to the requested simple numerical type:

```
e=Ens("c1ncccc1")
a=Atom.Ref(e,1)
print(int(a))
```

Ensembles provide only the boolean value check. It returns *True* if the ensemble has atoms, *False* otherwise:

```
if (e):
    print("Ensemble has atoms")
```

Reactions also have the boolean value check. It tests whether there are any ensembles in the reaction. In addition, the operator += and -= can be used to add or remove ensembles from a reaction:

```
xp=Reaction("C=C>>C(Br)C(Br)")
xp += (Ens("BrBr"),"agent")
xp -= xp[0]
```

Above code snippet first creates a reaction, then adds another ensemble with a defined role (see **`reaction add`** command), and finally the reagent ensemble is removed from the reaction.

Datasets have the same three numerical functions as reactions. The += method corresponds to **`dataset add`**.

The boolean test operator for molfiles tests that the file is neither at **`EOF`**, or in an error state. The += operator is a simple form of **`molfile write`**.

The boolean test for tables reports whether there are any rows in the table.

The boolean test for networks tests whether there are any vertices in the network.

Ensembles, reactions and datasets support the standard **`hash()`** function. The values are normal object hash values (`E_HASHISY`, `X_STEREO_ISOTOPE_HASH`, `D_HASHISY`) cast to the **`Py_hash_t`** type. On 32-bit platforms, this type has only 32 bits, so the hash value only contains the lower bits.

Ensembles support rich comparison operators, which test for structural identity using stereo/isotope hash codes and/or substructure match excluding hydrogens.

```
e1=Ens("CCC")
e2=Ens("CC")
e3=Ens("C(C)C")
if e2<=e1:
    print("is substructure or identical")
if e1==e3:
    print("structures topologically identical")
```

Both sample conditions match. While these comparison are convenient to write, they are pretty expensive if substructure matching is involved. In many cases, direct property data comparison is preferable for performance.

Copy and *Deepcopy* Support

Ensembles, reactions, datasets, molfiles, tables, networks and the custom bitvector object support the **`copy.copy()`** and **`copy.deepcopy()`** methods of the standard Python **`copy`** module via built-in **`__copy__`** and **`__deepcopy__`** methods. For toolkit objects, these two methods are functionally identical. Effectively, all copies of chemistry objects are deep. You always get full ensemble duplicates, not some ensemble body which refers to the same atoms and bonds as the source ensemble, which is an encoding scheme not supported by the core library.

Custom Object Attributes

In the TCL interface, the attributes which can be assigned to and read from objects is a fixed set. It is not possible to assign custom attributes. Property data is different - as long as there is a suitable property definition, any number of chemical properties can be attached to an object.

In the Python interface, the default behavior is the same. An error results if an attempt is made to read or write an undefined attribute. However, major chemistry reference objects carry a dictionary which can hold private attributes, if this is explicitly allowed. To enable this feature, set

```
cactvs[“custom_python_attributes”] = True
```

After that, statements like

```
e=Ens(“CCC”)
e.mydata = “TopSecret”
print(e.mydata)
```

will succeed. Since these custom attributes reside on the reference object, not the core library chemistry object, they are invisible from TCL. Standard predefined attributes will always have precedence, and it is not possible to use custom attribute names which look like property names.

Once a chemistry object has acquired a Python reference object, this object is re-used when a new reference object for the same chemistry object is requested while the first reference still exists, by simply incrementing is usage counter. Custom attributes are thus visible via all references to the same library object. However, once a reference object has gone completely out of scope, and all its usage counts decremented, custom attributes set on this item die with it. Requesting a new chemistry object reference thereafter instantiates a new reference object, with an empty custom attribute dictionary.

Python Computational Modules

It is possible to define properties which include a computational function written in Python. This works in the same way as the Tcl equivalent. When saved, the Python script source becomes a part of the property definition file. Property computations are executed in a per-property per-thread slave interpreter (in Python nomenclature, a sub-interpreter) and are thus cleanly isolated from any other property definition.

```
p=Prop('E_MYPROPERTY',datatype='color',functiontype='python',
    computefunction='CSgetE_MYPROPERTY',functionsource='''
def CSgetE_MYPROPERTY(eh):
    eh.E_MYPROPERTY = "orange"
''')
p.write()
e=Ens("c1ncccc1")
print(e.E_MYPROPERTY)
```

The property definition written out by the **`p.write()`** statement looks something like this (some standard attributes are omitted):

```
<?xml version="1.0" encoding="UTF-8" ?>
<!-- Property Definition for Cactvs Chemoinformatics Toolkit written 2019-02-04
19:16:29 -->
<property name="E_MYPROPERTY" objclass="ens" datatype="color"
classuuid="f11796e0-28a8-11e9-9f77-7379e9c1d6dd"
versionuuid="f11796e0-28a8-11e9-9f77-7379e9c1d6dd">
    <version>1.0</version>
    <date>2019-02-04 19:16</date>
    <author>Wolf-D. Ihlenfeldt</author>
    <license class="Proprietary"/>
    <invalidation>never</invalidation>
    <functions count="1">
          <function class="compute" name="CSgetE_MYPROPERTY" type="pythonscript"
count="1">
<![CDATA[
def CSgetE_MYPROPERTY(eh):
        eh.E_MYPROPERTY = "orange"
]]>
          </function>
    </functions>
</property>
```

Computable properties where the compute function is written in Python can be transparently used in a Tcl main script context, just like Tcl-computed properties are usable from Python.

However: There is one major exception to this. Python computational modules can only be used from the main thread. Python computation functions (or other property functions, such as data merging, verification, etc.) cannot be invoked from any secondary threads. This is due to the explicit non-support of sub-interpreters combined with multi-threading in the current standard Python runtime, regardless of GIL handling techniques. Still, a property which contains a computational Python function can still be referred to in secondary threads, as long as no Python function is implicitly or explicitly called. For example, the property may still be used as a simple definition template (setting name, data type, etc.) in other threads, so existing data may be recalled, or property data on objects may be manipulated directly by thread scripts, without calling the native computation function.

The same multi-threading limitations apply to any other Python function which could be invoked from multiple threads. More examples are multi-threaded file scanning with custom match functions, or dataset-associated object pipeline processing threads attached via the **`addthread`**() method.

Method Documentation

The Tcl commands and subcommands of the toolkit are documented side by side with the Python equivalent in a single document. Python method and function forms are written in dark gray, while the Tcl form is written in black. The Tcl form is always listed first.

Object methods start with a single letter, indicating the object type. Class methods start with the class name. Examples:

```
a.bonds(?filterset=?,?mode=?)
Atom.Ref(eref,identifier)
```

The first is an atom object method, while the second is an atom class method.

Most optional parameters are named in the Python interface. If optional parameter are not used, it is possible to skip them, usually by passing a **`None`** value. Alternatively, arguments after the skip position can be named. This is standard Python syntax.

```
a.bonds()
a.bonds(“carbon”)
a.bonds((“carbon”,”hydrogen”))
a.bonds(“!hydrogen”)
f=Filter.Ref(“carbon”)
a.bonds(f)
a.bonds(f,”exists”)
a.bonds(None,”count”)
a.bonds(mode=”count”)
```

These are all valid forms of calling the atom method to get references to the filtered or unfiltered bonds of an atom (or their counts, or an existence flag).

CACTVS Tcl and Python Scripting Language Reference

The *atom* Command

The *atom* command is the generic command used to manipulate atoms. The **Tcl** syntax of this command follows the standard schema of *command/subcommand/majorhandle/minorlabel.* For **Python**, most commands are object methods.

The pseudo atom labels *first* or ^, *last* or *end* or *$* and *random* are special values, which select the first atom in the atom list, the last, or a random atom.

Atoms can also be selected via atom indices (# prefix), atom mappings (^ prefix), previous labels (suffix *%*) or specific property values.

Examples:

```
atom get $ehandle 1 A_SYMBOL
atom get $ehandle ^1 A_SYMBOL
atom get $ehandle #0 A_SYMBOL
atom get $ehandle {A_LABEL = 1} A_SYMBOL
atom hadd $ehandle 2
```

This is the list of officially supported subcommands:

atom anchormatch

```
atom anchormatch ehandle label ss_ehandle ?ss_label? ?matchflags? ?ignoreflags?
   ?atommatchvar? ?bondmatchvar? ?molmatchvar?
```

```
a.anchormatch(substructure=,?substructureatom=?,?matchflags=?,?ignoreflags=?,
   ?atommatchvariable=?,?bondmatchvariable=?,?molmatchvariable=?)
```

This command is a variant of the **`atom match`** command. The difference is that the full substructure is matched, and not just its first or selected atom. A substructure match anchor between the command atom and the first or selected substructure atom is enforced (see **`-anchor`** option of the **`match ss`** command).

Example:

```
set eh [ens create CCO]
echo [atom match $eh 3 O(C)(C)]
echo [atom anchormatch $eh 3 O(C)C]
```

The first command matches, because only the first substructure atom is checked. The second fails, even though the first substructure atom is a match - but then its environment does not fit.

atom angle

```
atom angle ehandle label label2 label3 ?property?
```

```
a.angle(atom2=,atom3=,?coordinateproperty=?)
```

Compute the angle between 3D atomic coordinates stored in a property between the three atom arguments, which are considered linked in the specified sequence. The source property for atomic coordinates is by default `A_XYZ`, but another property can be set, which also needs to be an atomic float vector.

The return value is the angle in degrees between the vectors implicitly constructed from the 3D atomic coordinate of the second atom pointing to that of the first, and from the second atom to the third. No bonds need to exist between the atoms. All atoms used in a statement must be different, and possess 3D coordinates initially, or after an automatically started computation of the source property.

atom append

```
atom append ehandle label ?property value?...
a.append({?property:value,?...})
a.append(?property,value,?...)
```

Standard data manipulation command for appending property data. It is explained in more detail in the section about setting property data.

The command returns the first data value.

Example:

```
atom append $ehandle 1 A_SUPERATOMSTRING "_linker"
```

atom atom

```
atom atom ehandle identifier
Atom.Ref(eref,identifier)
```

Standard cross-referencing command to obtain the label (or reference object, for **Python**) of the atom as stored in property `A_LABEL`. This is explained in more detail in the section about object cross-references.

Example:

```
atom atom $ehandle #0
```

returns the label of the first atom of the ensemble atom list.

atom bondangles

```
atom bondangles ehandle label ?filterset? ?filtermode?
a.bondangles(?filterset?,?mode?)
```

Standard cross-referencing command to obtain the labels or references of the bond angle objects the atom is participating in. This is explained in more detail in the section about object cross-references.

atom bonds

```
atom bonds ehandle label ?filterset? ?filtermode?
a.bonds(?filterset?,?mode?)
```

Standard cross-referencing command to obtain the labels or references of the bonds the atom is participating in. This is explained in more detail in the section about object cross-references.

Examples:

```
atom bonds $ehandle 1
atom bonds $ehandle 1 {1 doublebond triplebond} count
```

The first example returns all labels of the bonds atom 1 is participating in. The second example returns the number of double or triple bonds the atom is a part of.

atom change

```
atom change ehandle label element ?linkatom? ?removeh?
a.change(replacement=,?linkatom=?,?removeh=?)
```

This command is very similar to the command **`atom replace`**. The important difference is that the *element* parameter is always interpreted as an element symbol encoding, and not primarily as an ensemble handle, ensemble handle/molecule label pair or **SMILES** string.

The rest of the command is explained in the paragraph on **`atom replace`**.

Example:

```
atom change $eh 1 C
atom change $eh 2 Z
```

The first example changes the atom with label 1 to a neutral carbon atom. Bonds of the old atom 1 are inherited if possible. If this is not possible due to valence violations, an error is raised. The second example changes an atom to a query specification for an electro-negative element.

atom charge

```
atom charge ehandle label chargedelta
a.charge(chargedelta=)
```

Try to change the formal charge `A_FORMAL_CHARGE` of the atom by the specified amount. The free electron count `A_FREE_ELECTRONS` is also adjusted, and other charge- or free-electron-dependent properties on the ensemble are recursively invalidated. Impossible final charge values are rejected. If the desired charge state can only be reached by deprotonation, this is automatically attempted, and a bond change property invalidation event is processed.

The command returns the atom label (for **TCL**) or reference (for **PYTHON**).

atom create

```
atom create ehandle ?symbol? ?bondtype atomlabel?...
Atom(eref,?symbol?,?bondtype,aref?...)
Atom.Create(eref,?symbol?,?bondtype,aref?...)
```

Create a new atom in the ensemble. By default, the atom is added without any bonds or charge and the standard set of free electrons. The *symbol* parameter is usually an element symbol, which is decoded in a case-sensitive fashion. If it is omitted, an *unspecified* atom is created. The isotopic element symbols **D** and **T** are recognized and decoded to the corresponding hydrogen isotopes, setting the `A_ISOTOPE` property.

This command may also be used to add various pseudo and query atoms. Allowable symbols for this purpose are

- **3DPOINT** or **DU** or **BQ** for points in 3D space
- **POLY** for polymers
- **EPAIR**, **EP** or **LP** for lone pairs
- * or **OV** for an open valence pseudo atom
- ~ for a superatom with a yet undefined identifier string

- **HA** for a generic hydrogen acceptor
- **HD** for a generic hydrogen donor
- **A** for a query atom which is not hydrogen
- **Q** for a query atom which is a hetero atom (not C or H)
- **M** for a query atom which is a metal
- **?** for a query atom which can be any atom
- **X** for a query element list with the halogens
- **Y** for a query element list with the electro-negative elements N,O,Cl,Br
- **Z** for a query element list with the electro-negative elements N,O,F,S,Cl,Br,I
- **L** for a query element list with a yet undefined set of elements
- **@** for a delocalization anchor
- **R** for a query atom of type *insulator*.

Instead of an element symbol, the periodic system number of an element may also be used, optionally prefixed with a hash character (#) in **SMILES** style. Additionally, the standard **BEILSTEIN** query atoms, such as '**[Alk]**', as well as CCDC element groups, such as '**[3A]**', are supported.

If the superatom symbol ~ is followed by more characters, these are copied to the superatom identifier string (`A_SUPERATOMSTRING` property). If a known fragment is specified this way, it may be expanded later.

The command returns the automatically assigned label of the new atom, or the atom reference for **PYTHON**. Note that this command does not require a label parameter, since it creates new atoms.

This command updates the ensemble information and recursively purges information which is susceptible to atom changes. For atom properties which survive this step, a default value is added, if the property is not part of the set of properties managed actively by this command, such as the free electron count and the atom label.

After the atom symbol, an additional sequence of (non-nested) bond type and atom label parameters may be specified. The recognized bond types are the same as in **bond create**. Bonds of the requested type are created and link the new atom to existing atoms in one step. This bond creation process is limited by the valence restrictions of the involved atoms. Successful bond creation triggers a bond change property data invalidation event.

The **atom create** command can also be accessed, for historical reasons, as **atom add**. This alias is deprecated.

The return value is the label of the new atom.

Examples:

```
atom create $ehandle C
atom add $ehandle ?
atom expand $ehandle [atom add $ehandle ~FMOC]
atom create $ehandle N = $a1 single $a2
```

The first example adds a carbon atom to the ensemble. The second line adds an *any* query pseudo atom, which, in the context of a substructure search, matches any atom. The third example adds a superatom named *FMOC* in the inner command. Since this is a fragment name the library understands by default, it may be expanded to the full *FMOC* fragment with the outer command. Finally, a nitrogen atom is added and immediately bonded via a double bond to the atom identified by the label in variable **`a1`**, and via a single bond to the atom in **`a2`**.

atom defined

```
atom defined ehandle label property
a.defined(property)
```

This command checks whether a property is defined for the atom. This is explained in more detail in the section about property validity checking. Note that this is *not* a check for the presence of property data! The **`ens valid`** command is used for this purpose.

Example:

```
atom defined $ehandle 1 A_XYZ
```

checks whether atom 1 is of a type for which `A_XYZ` is defined.

The command returns a boolean status value.

atom delete

```
atom delete ehandle ?label?...
atom delete ehandle all
a.delete()
Atom.Delete(eref,?alabel/aref/arefsequence?,...)
Atom.Delete(aref,...)
Atom.Delete(eref,"all")
```

Delete zero or more atoms. All bonds which the atoms participate in are also deleted. The electron counts of surviving atoms participating in deleted bonds are automatically updated. Molecule and ring information, and other minor object classes under the control of the ensemble major object which depend on an unchanged atom set are deleted. Any property data which depends on an unchanged atom set is also invalidated, or, if the property is set up to do so, re-computed.

Note that this command does not delete hydrogen atoms the deleted atoms were bonded to. These remain in the ensemble as isolated, now unbonded atoms. The **`atom xdelete`** subcommand also deletes these hydrogen atoms.

The special atom label *all* requests deletion of all atoms. Usually, this is equivalent to **`ens clear`**.

The return value of the command is the number of deleted atoms.

Example:

```
atom delete $ehandle 1
```

This command is one of few atom subcommands which do not require an atom label. If no label is given, the command does nothing. This is useful for list expansions where the list might be empty:

```
eval atom delete $ehandle $delatomlist
atom delete $ehandle {*}$delatomlist
```

atom dget

```
atom dget ehandle label propertylist ?filterset? ?parameterdict?
a.dget(property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **`atom get`** command. The difference between **`atom get`** and **`atom dget`** is that the latter does not attempt computation of property data, but rather initializes the property values to the default and return that default if the data is not yet available. For data already present, **`atom get`** and **`atom dget`** are equivalent.

atom deprotonate

```
atom deprotonate ehandle label ?count?
a.deprotonate(?count=?)
```

Attempt to remove one ore more protons from the atom, with adjustment of formal atom charge and processing of appropriate structure change property data invalidation events.

The command returns the atom label (for **TCL**) or atom reference (for **PYTHON**).

atom distance

```
atom distance ehandle label ?label2? ?property?
a.distance(?atom2=?,?coordinateproperty=?)
```

Compute the 3D distance between two atoms based on the values of a coordinate property. The source property for atomic coordinates is by default `A_XYZ`, but another property can be specified, which also needs to be an atom float vector.

The command returns the value as a floating point number in the unit of the source property (Angstrom in case the default `A_XYZ` is used). An equivalent explicit vector arithmetic script is

```
vec len [vec subtract [ens get $eh $label A_XYZ] [ens get $eh $label2 A_XYZ]]
```

If a second atom identifier is not specified, or given as an empty string, the result is a nested list of the distances to all bonded neighbor atoms, regardless of the bond types. Each sublist consists of the partner atom label and the bond length from the current atom to that neighbor.

In order to obtain the topological distance between atoms, use the **`atom topodistance`** command, or compute property `A_TOPO_DISTANCE`.

atom dup

```
atom dup ehandle ?label_list? ?datasethandle? ?position?
a.dup(?dataset=?,?position=?)
Atom.Dup(eref,aref_tuple,?dataset=?,?position=?)
```

Duplicate zero or more atoms, plus all the bonds existing between them, into a new ensemble. This command is very similar to **`ens fragment`**, and the same caveats about preserved and destroyed data in the duplicate apply.

By default, the new ensemble is appended to the same dataset as the original ensemble, or placed outside of any dataset if the input ensemble was not a dataset member. If the optional dataset handle parameter is specified, the duplicate is directly moved to that dataset. If an empty string is passed, the duplicate is not made a dataset member, even if the input ensemble is in a dataset.

If the duplicate is moved to a dataset, it is appended to the dataset end by default. This happens also if the position parameter is explicitly specified as *end* or an empty string. Otherwise, the ensemble is inserted at the given position, starting with 0. If the requested position is larger than the current size of the dataset, the ensemble is appended.

The command returns the handle of the new ensemble object for **Tcl**, or an ensemble reference for **Python**.

Example:

```
set ehfrag [atom dup $eh {*}$alist]
```

atom ens

```
a.ens()
```

Python-only method to get the ensemble reference from an atom reference.

atom exists

```
atom exists ehandle label ?filterlist?
a.exists(?filters=?)
Atom.Exists(eref,label,?filters=?)
```

Check whether this atom exists. Optionally, a filter list can be supplied to check for the presence of specific features. The command returns 0 if the atom does not exist, or fails the filter, and 1 in case of successful testing.

Example:

```
atom exists $ehandle 99
```

atom expand

```
atom expand ehandle label ?allowambiguous? ?noimplicith?
a.expand(?allowambiguous=?,?noimplicith=?)
```

This command attempts to expand a superatom. A superatom is either an atom for which the atom type property `A_TYPE` is set to *super* (the preferred method), or a standard atom (`A_TYPE` *normal*) with certain property data.

For a successful expansion, the first class of explicit superatoms must have a valid `A_SUPERATOMSTRING` property value which can be located in the table of known superatom identifiers. The second class of normal atoms needs a valid `A_TEXTLABEL` property data with a known superatom identifier in its *label* text field. The use of normal atoms as superatom surrogates is deprecated.

If the *allowambiguous* flag parameter is set, superatoms of uncertain status are expanded. Some superatom names are ambiguous, for example *Al*, which may both refer to the element and *alanine*. The superatom table protects against unchecked expansion of such atoms by containing an ambiguity flag which is set in such cases.

By default, the fragments from the superatom table are imported with a full set of hydrogens. If the optional *noimplicith* flag is set, only hydrogens which are explicitly spelled out in the superatom definition are included. For example, superatoms *COO* and *COOH* are expanded to the same form with an acidic hydrogen by default, but if the flag is set, only the second form has it.

The command returns 1 if the atom was a superatom and expansion was successful, 0 otherwise. It may also raise an error if a superatom was found, but expansion failed, for example because of an illegal bonding situation which does not allow the creation of the required normal bonds to the expanded fragments.

The expanded superatom and all other atoms in the original ensemble retain their labels.

Only a single level of superatoms is expanded - if the expanded fragment contains another superatom, it remains in its original form.

Examples:

```
atom expand $ehandle [atom create $ehandle ~BOC]
```

This command immediately expands the freshly created BOC fragment. A command sequence like

```
atom set $ehandle [ens create C 0] A_TEXTLABEL(label) COOMe
atom expand $ehandle 1
```

also works, but is deprecated.

atom expr

```
atom expr ehandle label expression
a.expr(expression)
```

Compute a standard **SQL**-style property expression for the atom. This is explained in detail in the chapter on property expressions.

atom fill

```
atom fill ehandle label ?property value?...
a.fill({property:value,...})
a.fill(?property,value?,...)
```

Standard data manipulation command for setting data, ignoring possible mismatches between the lengths of the lists of objects associated with the property and the value list. It is explained in more detail in the section about setting property data.

Example:

```
atom fill $ehandle 1 B_COLOR red
```

sets the color of the first bond atom 1 participates in to *red*.

The command returns the first fill value.

atom filter

```
atom filter ehandle label filterlist
a.filter(filters)
```

Check whether an atom passes a filter list. The return value is boolean 1 for success and 0 for failure.

Example:

```
atom filter $ehandle 1 [list carbon doublebond]
```

checks whether the atom is a carbon atom with a double bond.

atom get

```
atom get ehandle label propertylist ?filterset? ?parameterdict?
a.get(property=,?filters=?,?parameters=?)
a[property]
a.property
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

Examples:

```
atom get $ehandle 1 {A_SYMBOL A_ELEMENT}
```

yields the atomic symbol and the element number of atom 1 as a list. If the information is not yet available, an attempt is made to compute it. If the computation fails, an error results.

```
atom get $ehandle 1 B_ORDER ringbond
```

will give the bond orders of all bonds of the atom which are ring bonds.

For the use of the optional property parameter list argument, refer to the documentation of the **`ens get`** command.

Variants of the **`atom get`** command are **`atom new, atom dget, atom nget, atom show, atom sqldget`**, **`atom sqlget, atom sqlnew`** and **`atom sqlshow.`**

Further examples:

```
atom get $ehandle 1 A_SYMBOL
atom get $ehandle 1 A_FLAGS(boxed)
```

In the Python case, the first variant accepts property lists/tuples containing string property names and/or property references, or a string property list in addition to a single property. Property references can be used instead of strings only in the first variant, both as single arguments or as part of lists/tuples. Direct indexed access to property fields also requires the first version, as does the use of filters or specific computation parameters.

atom groups

```
atom groups ehandle label ?filterset? ?filtermode?
a.groups(?filters=?,?mode=?)
```

Standard cross-referencing command to obtain the labels or references of the groups the atom is a member of. This is explained in more detail in the section about object cross-references.

Example:

```
atom groups $ehandle 1
```

atom hadd

```
atom hadd ehandle label ?filterset? ?flags? ?chargedelta?
a.hadd(?filters=?,?flags=?,?chargedelta=?)
```

Add a standard set of hydrogens to the atom. If the *filterset* parameter is specified, the atom needs to pass the filter set in order to be processed.

Additional operation flags may be activated by setting the *flags* parameter to a list of flag names, or a numerical value representing the bit-ored values of the selected flags. By default, the flag set is empty, corresponding to the use of an empty string or *none* as parameter value. These flags are currently supported:

- *no2dcoords*
  Do not assign 2D coordinates to the added hydrogens, even if the rest of the atoms in the ensemble have valid 2D coordinates. In any case, 2D coordinates are never added when the ensemble does no already possess valid 2D coordinates.
- *no3dcoords*
  Do not assign 3D coordinates to the added hydrogens, even if the rest of the atoms in the ensemble have valid 3D coordinates. In any case, 3D coordinates are never added when the ensemble does no already possess valid 3D coordinates.
- *noanions*
  Do not add hydrogen to atoms with a negative formal charge.
- *noatoms*
  Do not add hydrogen to atoms without any bonds.
- *nocations*
  Do not add hydrogen to atoms with a positive formal charge.
- *noelements*
  Do not add hydrogen if the ensemble consists purely of isolated metal atoms, which probably represent the material in elementary form, or as an alloy.
- *noexcessvalences*
  Similar to *nohighvalences*, but hydrogen is not added to any atom which is not in its lowest standard bonded valence state.
- *nofixatomtext*
  Do not adjust property `A_TEXTLABEL` (if present) by removing references to implicit H from it on atoms where hydrogen is added. For example, by default “NHCOOEt” becomes “NCOOEt” after adding an instantiated hydrogen to the nitrogen atom. This reduces confusion on the hydrogen status when rendering all atoms.
- *nohighvalences*
  Do not add hydrogen to atoms which already exceed their lowest standard valence minus any formal charge. This option only applies to elements which have a defined lowest standard valence (this is configurable via the element table).
- *nomemory*
  Do not remember the added hydrogen atoms as automatically added. Normally, a flag is retained as part of the atom information which distinguishes atoms which were added by automatic processing, such as hydrogen addition, from those which were originally input.
- *nometals*
  Do not attempt to add hydrogen to atoms which are metals (as defined in the system element table).
- *nospecial*
  Do not perform hydrogen addition to atoms which participate in non-standard bonds (all bonds with `B_TYPE` not *normal*).
- *keepflags*
  For expert use only. Do not discard min/max values and property scope flags for atom properties when hydrogen is added.

- *protonate*
  Add a single proton to the atom. The charge of the atom is increased, only a single hydrogen is added regardless of the standard number of missing hydrogens, and this command *will* issue the standard property invalidation event for atom and bond changes.
- *resetmemory*
  Reset the origin flag described above for all atoms in the ensemble. All current atoms appear to be part of the original atom set.

If a charge delta parameter is specified, the atomic charge and free electrons of the atom are adapted accordingly before the hydrogens are added. The manipulation of the charge usually changes the number of added hydrogen atoms. It is not possible to change the charge in such a way that the number of free electrons would become negative.

Adding hydrogens with this command, except if the *protonate* flag is set, is less destructive to the property data set of the ensemble than adding them with individual **`atom create/bond create`** commands, because many properties are designed to be indifferent to explicit hydrogen status changes, but are invalidated if the structure is changed in other ways.

The command returns the number of hydrogens which were added.

Example:

```
set ehandle [ens create FC(F)(F)(F)]
atom delete $ehandle 1
atom hadd $ehandle 2
```

transforms *tetrafluoromethane* to *trifluoromethane*.

atom hdelete

```
atom hdelete ehandle ?label?...
atom hdelete ehandle all
a.hdelete()
Atom.Hdelete(eref,?alabel/aref/arefsequence?,...)
Atom.Hdelete(aref,...)
Atom.Hdelete(eref,“all”)
```

Delete zero or more atoms. All bonds which the atoms participate in are also deleted. The electron counts of surviving atoms participating in deleted bonds are automatically updated. Molecule and ring information, and other minor object classes under the control of the ensemble major object which depend on an unchanged atom set are deleted. Any property data which depends on an unchanged atom set is also invalidated, or, if the property is set up to do so, re-computed.

Additionally, and different from the simple **`atom delete`** command, all cut VB valences on the neighbor atoms which will not be deleted with the same statement are saturated with added hydrogen atoms. Only the cut valences are treated, this is not necessarily equivalent to a **`atom hadd`** command on the neighbors.

Otherwise, the command performs the same actions as the simple **`atom delete`** command.

The *all* command variants are identical to that of the simple **`atom delete`** command since no neighbor atoms for hydrogenation remain.

The command returns the number of deleted atoms.

atom hdup

```
atom hdup ehandle ?label_list? ?datasethandle? ?position?
a.hdup(?dataset=?,?position=?)
Atom.Hdup(eref,aref_tuple,?dataset=?,?position=?)
```

Duplicate zero or more atoms, plus all the bonds existing between them, into a new ensemble, and plug all open valences by adding standard hydrogens. This command is similar to **`ens hfragment`**, and the same caveats about preserved and destroyed data in the duplicate apply.

By default, the new ensemble is appended to the same dataset as the original ensemble, or placed outside of any dataset if the input ensemble was not a dataset member. If the optional dataset handle parameter is specified, the duplicate is directly moved to that dataset. If an empty string is passed, the duplicate is not made a dataset member, even if the input ensemble is in a dataset.

If the duplicate is moved to a dataset, it is appended to the dataset end by default. This happens also if the position parameter is explicitly specified as *end* or an empty string. Otherwise, the ensemble is inserted at the given position, starting with 0. If the requested position is larger than the current size of the dataset, the ensemble is appended.

The command returns the handle of the new ensemble object for **Tcl**, or an ensemble reference for **Python**.

atom hstrip

```
atom hstrip ehandle label ?flags? ?chargedelta?
a.hstrip(?flags=?,?chargedelta=?)
```

This command removes hydrogens from the selected atom. By default, all hydrogen atoms are removed.

The *flags* parameter can be used to make the operation more selective. It may be a list of the following flags:

- *deprotonate*
  If this flag is set, a single proton is removed from the atom. This command variant *does* issue a standard atom and bond change property invalidation event, and it always ends processing after removing the first proton. Proton removal decreases the charge of the atom by one.
- *keepalphawedge*
  Keep hydrogen atoms which are bonded to an atom which is at the tip of a wedgebond. This flag excludes the case where the bond to the hydrogen atom is the wedge bond - use the *keepwedge* flag to cover this case.
- *keepisotopes*
  Keep hydrogen atoms which are isotope labels (including enriched/depleted ^{1}H).
- *keeporiginal*
  Hydrogen atoms which were not automatically added via a hydrogen addition command are retained. Note that these commands can be run in a mode which does not leave information about automatic addition - hydrogens added this way do not survive.
- *keepprotons*
  Keep any molecules which consist only of hydrogen atoms (such as protons, hydride anions, and molecular hydrogen).

- *keepspecial*
  If this flag is set, hydrogens which are usually displayed, such as on aldehydes, wedge bonds, carbon triple bonds or hetero atoms are retained.
- *keepwedge*
  Keep hydrogens which are at the end of a wedge bond, indicating stereochemistry.
- *normalize*
  Normalize the wedge pattern for standard cases, removing excess wedges from hydrogens if the result structure is still stereochemically defined. Hydrogens which lose their wedge in this process are no longer protected by the *keepwedge* flag.
- *wedgetransfer*
  If a hydrogen atom is removed which is at the end of a wedge, the wedge information is saved by transferring the wedge (changing its up/down status if necessary) to an adjacent, surviving bond. This flag has no effects if the *keepspecial* or *keepwedge* flags are set. This flag is set by default.

If the *flags* parameter is an empty string, or *none*, it is ignored. The default flag value is *wedgetransfer* - but the default value is overridden if any flags are set!

If a charge delta parameter is specified, the atomic charge and free electrons of the atom are adapted accordingly before the hydrogens are added. The manipulation of the charge will usually change the number of added hydrogen atoms. It is not possible to change the charge in such a way that the number of free electrons would become negative.

Hydrogen stripping is not as disruptive to the ensemble data content as normal atom deletion, except in case the *deprotonate* flag is set. The system assumes that this operation is done as part of some file output or visualization preparation. However, if any new data is computed after stripping, the computation functions see the stripped structure, and proceed to work on that reduced structure without knowledge that the structure may contain implicit hydrogens.

The return value of the command is the number of hydrogens removed.

Example:

```
atom hstrip $ehandle 1 [list keeporiginal wedgetransfer]
```

atom hydrogenate

```
atom hydrogenate ehandle label ?filterset? ?changeset?
a.hydrogenate(?filters=?,?changeset=?)
```

Reduce all bonds the atom participates in to single bonds except those excluded by the filter set.

If a change set is supplied, its interpretation is the same as in **`atom hadd`**.

The command returns the number of added hydrogens.

Example:

```
atom hydrogenate $eh 1 {!arobond !ccbond}
```

This reduces all non-aromatic hetero bonds atom 1 participates in to single bonds.

atom index

```
atom index ehandle label
```

```
a.index()
```

Get the index of the atom. The index is the position in the atom list of the ensemble. The first position is index 0.

Example:

```
atom index $ehandle 99
```

atom invert

```
atom invert ehandle label
a.invert()
```

Invert the stereochemistry at the atom, provided it is an sp3-type atomic stereo center, which includes those which use an electron pair as pseudo ligand and allenes with an odd number of atoms. This command updates any atomic stereo descriptors and bond wedges to the ligands if set, but only compute `A_LABEL_STEREO`. No check it made whether the atom can physically be a stereo center, but if the `A_LABEL_STEREO` descriptor is zero, or describes non-sp3 types of stereochemistry such as square planar, the command does nothing and returns 0, but will not raise an error. For odd allenes, bond wedges at the terminal atoms are updated, not those at the center atom.

If stereochemistry was inverted, this command issues a stereo change property invalidation event and additionally invalidates the `A_STEREOGENIC` and `B_STEREOGENIC` properties, because the stereo potential of centers which possess two ligand groups which only differ in stereochemistry may have changed.

If the command finds a defined stereo center and succeeds in inverting it, it returns 1, 0 otherwise.

atom isotopecheck

```
atom isotopecheck ehandle label ?extended?
a.isotopecheck(extended=)
```

Test whether the isotope label of the atom, if it exists, is physically reasonable. The command returns boolean *true* if the label is OK, or is not set. If no isotope is set in `A_ISOTOPE`, the command always reports no problems.

By default, a smaller isotope table is used which contains only isotopes which are sufficiently long-lived to perform chemistry on. These include naturally occurring isotopes as well as isotopes used for experimental labeling, such as ^{1}T or ^{14}C. If the *extended* boolean flag is set, a larger table containing all known isotopes of the elements is used.

The *isocheck* command is an alias.

atom jget

```
atom jget ehandle label propertylist ?filterset? ?parameterdict?
a.jget(property=,?filters=?,?parameters=?)
```

This is a variant of **`atom get`** which returns the result data as a **JSON** formatted string instead of **Tcl** or **Python** interpreter objects.

atom jnew

```
atom jnew ehandle label propertylist ?filterset? ?parameterdict?
a.jnew(property=,?filters=?,?parameters=?)
```

This is a variant of **`atom new`** which returns the result data as a **JSON** formatted string instead of **TCL** or **PYTHON** interpreter objects.

atom jshow

```
atom jshow ehandle label propertylist ?filterset? ?parameterdict?
a.jshow(property=,?filters=?,?parameters=?)
```

This is a variant of **`atom show`** which returns the result data as a **JSON** formatted string instead of **TCL** or **PYTHON** interpreter objects.

atom local

```
atom local ehandle label propertylist ?filterset? ?parameterdict?
a.local(property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading and recalculating object data. It is explained in more detail in the section about retrieving property data.

Example:

```
atom local $ehandle 1 A_LABEL_STEREO
```

Note that very few computation routines currently support the local re-computation of data - in most cases, this command falls back to in a global re-computation.

atom match

```
atom match ehandle label ss_ehandle ?ss_label? ?matchflags? ?ignoreflags?
   ?atommatchvar? ?bondmatchvar? ?molmatchvar?
a.match(substructure=,?substructureatom=?,?matchflags=?,?ignoreflags=?,
   ?atommatchvariable=?,?bondmatchvariable=?,?molmatchvariable=?)
```

Check whether the selected atom matches a substructure. Only the first substructure atom, or the atom selected by the substructure label parameter, is tested. The substructure may be part of any structure ensemble, and even be in the same ensemble as the primary command atom.

The precise operation of the substructure match routine can be tuned by providing a standard set of match flags and feature ignore flags. The default match flag set has set bits for the *bondorder*, *atomtree* and *bondtree* comparison features, and an empty ignore set. If a flag set is specified as an empty string, the default set is used. In order to reset the flag set, an explicit *none* value must be used. The bit options of the match flag are explained in the documentation of the **`match ss`** command.

The command returns 1 for a successful match, 0 otherwise. If an optional atom, bond, or molecule match variable is specified, it is set to a nested list of matching substructure/structure atom, bond or molecule labels (references for **PYTHON**). If no match can be found, the variable is set to an empty list. In case only a bond or molecule match variable is needed, an empty string can be used to skip the unused match variable argument positions.

Example:

```
set ss [ens create {[F,Cl,Br,I]} smarts]
set a_is_halogen [atom match $ehandle $label $ss 1]
```

atom mol

```
atom mol ehandle label ?filterset? ?filtermode?
a.mol(?filters=?,?mode=?)
```

Standard cross-referencing command to obtain the label (for **Tcl**) or reference (for **Python**) of the molecule the atom is a member of. This is explained in more detail in the section about object cross-references.

Examples:

```
atom mol $ehandle 1
atom mol $ehandle 1 heterocycle
```

The first example returns the label of the molecule. Note that it is possible for pseudo atoms to be outside of any molecule. In this case, an empty string is returned. The second example returns the molecule label if the atom is part of a molecule which contains one or more heterocycles. If the molecule does not contain a heterocycle, an empty string is returned. Note the use of *mol* in singular - an atom can only be a member of one molecule, or of none.

atom neighbors

```
atom neighbors ehandle label ?filterset? ?filtermode? ?sphere? ?allowduplicates?
a.neighbors(?filters=?,?mode=?,?sphere=?,?allowduplicates=?)
```

This command (which can also be invoked as subcommand *neighbours*, or *ligands*) is a cross-referencing command with some extra options and, in some filter modes, slightly different behavior than the standard object cross-reference subcommands.

In the simplest case, it returns the labels (for **Tcl**) or references (for **Python**) of the immediate neighbor atoms. A neighbor atom is an atom which is bonded via a standard (covalent, `BTYPE_VB`) or complex (`BTYPE_COMPLEX`) bond to the originating atom. In case the filter list contains bond filters, the bond leading to the originating atom must pass the check, not just any bond of the neighbor atom.

Example:

```
atom neighbors $ehandle 1 doublebond
```

returns all neighbor atom labels which are *bonded* via a double bond. Neighbor atoms which participate in a double bond with other atoms, but not the originating atom, are not returned.

This command supports special *filtermode* parameters in addition to the standard set (*exists*, *count*, *exclude*, *include*). The *notraverse* parameter, followed by a list of atom labels in any of the standard atom specification styles is a list of atoms which are not traversed during sphere expansion. The *bonds* parameter, followed by a bit set combination from the allowed values *ring*, *sidechain* or *bridge* can be used for topological filtering of the traversable bonds. By default, no topological bond filtering is applied.

Example:

```
atom neighbors [ens create CC(C)C] 2 {} {notraverse {3 4}} 2
```

only returns the hydrogen atoms 5, 6, 7 on atom 1, since carbon atoms 3 and 4 are blocked. If the atoms in the traversal block list are part of the requested sphere, they are listed.

By default atoms in the immediate neighborhood are examined, but this change be changed by the *sphere* parameter. The immediate neighbors are in sphere 1 (the default for this parameter), the next group of atom is in sphere 2, and so on. If the sphere is not 1, the special filtering of bonds is no longer active and the normal object substitution mechanism for cross referencing is used. When going beyond the first sphere, it is also possible that an atom may be reached by multiple paths of the selected length. By default, these atoms are returned only once, but with the last optional parameter this behavior may be changed. A positive sphere value only selects atoms in that sphere. A negative sphere parameter value returns a list of all neighbors up to and including the sphere identified by the absolute sphere value.

Example:

```
atom neighbors $ehandle 1 {carbon aroatom} count 2
```

counts the number of aromatic carbon atoms in a distance of two bonds.

atom new

```
atom new ehandle label propertylist ?filterset? ?parameterdict?
a.new(property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **`atom get`** command. The difference between **`atom get`** and **`atom new`** is that the latter forces the re-computation of the property data, regardless whether it is present and valid, or not.

atom nget

```
atom nget ehandle label propertylist ?filterset? ?parameterdict?
a.nget(property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **`atom get`** command. The difference between **`atom get`** and **`atom nget`** is that the latter always returns numeric data, even if symbolic names for the values are available.

atom paths

```
atom paths ehandle label targetlabel ?minlength? ?maxlength? ?filterset?
   ?atomproperty? ?maxpathcount? ?flags?
a.paths(target,?minlength=?,?maxlength=?,?filters=?,?atomproperty=?,
   ?maxpathcount=?,?flags=?)
```

This command finds all paths between a pair of atoms, walking along bonds of the types which define molecules. By default, these are bond types *normal*, *complex* and *3center*, but this can be changed by modifying the control variable *::cactvs(molecule_bond_set)*.

The return value of the command is a nested list, even it only a single path is found. Every sublist contains all the labels (for TCL) or references (for PYTHON) of the atoms in a single path, including those of the start and end atoms. Every bond is used only once in any path, and no path crossings through an atom are allowed. Every atom, with the possible exception of path end points, appear only once in any single path. Paths from an atom via some bonds back to itself are allowed. The atom must be a ring member for such paths to exist.

If the destination atom is specified as an empty string, all possible paths emerging from the source atom and not violating any other specified constraints are returned. This includes shorter sub-paths which are contained in a longer paths - these are reported as separate result items.

By default, all paths of length greater than zero are returned. The lengths of acceptable paths may be specified by the optional parameters. If only the minimum length is set, this value is also used for the maximum length, resulting in only paths of a specific length to be reported. The maximum path count parameter can be used to limit the number of paths found. However, the order of the found paths does depend on the arrangement of the atoms in the bonds, so it is generally not canonic. Omitting this parameter or setting it to a negative value disabled the maximum path count check.

A non-empty filter set can be used to restrict the atoms that are eligible to be part of the path. Normally, these are atom filters, such as *!hydrogen*, but other types may be used in special circumstances. Bond filters are however applied to the union of all bonds of an atom, not just the specific bond traversed in a path. For example, a *doublebond* filter lets an atom pass if it participates in any double bond, and does not necessarily mean the bond the atom was reached over in the path. Filters are not applied to the start atom of the path.

The default report value for an atom is its label, i.e. property `A_LABEL`. However, any other present or computable atom property may be specified instead with the optional atom property parameter. The parameter may also refer to a property field in case the property is indexible.

The final optional flag parameter is a list of additional keywords which further modify the path atom selection and result reporting. Currently, the following keywords are recognized:

- *noringchaincrossing*
  The path may not jump from a chain atom to a ring atom, or vice versa
- *concatenate*
  The report format for each individual path is not a Tcl list, but a string where the report atom property values are directly concatenated
- *printbondorder*
  Every report value after that of the first path atom is prefixed with a character indicating the bond order from the set *"-=#&"* for bond orders one to four, with a colon for aromatic bonds, and a question mark for non-VB bonds. Additionally, a @ is added if the bond closes a ring to the first path atom.

Example:

```
atom paths [ens create C1CCC1] 1 1
```

reports the paths {1 2 3 4 1} and {1 4 3 2 1}, which correspond to walking the ring clockwise and counter clockwise, respectively.

```
atom paths [ens create CC=C] 1 3 3 3 {} A_ELEMENT -1 {printbondorder concatenate}
```

returns {6-6=6}.

atom pis

```
atom pis ehandle label ?filterset? ?filtermode?
a.pis(?filters=?,?mode=?)
```

Standard cross-referencing command to obtain the labels or references of the π systems the atom is a member of. This is explained in more detail in the section about object cross-references.

Examples:

```
atom pis $ehandle 1
```

Get the labels of the π systems the atom is participating in. π systems are a rather exotic feature and not commonly used. These are essentially descriptions of bonding interactions which use p or d orbitals, such as in standard covalent multiple bonds. A simple double bond is described with one σ system and one π system in this representation.

atom protonate

```
atom protonate ehandle label ?count?
a.protonate(?count=?)
```

Attempt to add one ore more protons to the atom, with adjustment of formal atom charge and processing of appropriate structure change property data invalidation events.

The command returns the label (for **Tcl**) or reference (for **Python**) of the atom.

atom purge

```
atom purge ehandle label propertylist/stereo/isotope/query
a.purge(propertylist/stereo/isotope/query)
```

Reset existing property data on an atom. In case the argument is a list of property names, the value on that atom only is reset to the default value of the property. In case the property is not present on the ensemble, the command is ignored. The reset via a property list does *not* trigger a property dependency update. If that is desired, an **ens taint** command must be explicitly scripted.In case a reset property is a bond property instead of an atom property, the reset is executed for all bond atoms. Other property object class mismatches are currently not supported.

In addition to standard properties, several special pseudo property names are recognized.

The *stereo* code resets all atom-centered stereo information on the atom, including wedges in property B_FLAGS that point to the atom, and will trigger a stereo change event on the ensemble which may invalidate additional data.

The *isotope* code resets property A_ISOTOPE on the atom, marks the isotope data as tainted and runs a data dependency check.

The *query* code resets property A_QUERY, marks the query data as tainted and runs a data dependency check.

The command returns the label (for **Tcl**) or reference (for **Python**) of the atom.

atom ref

```
Atom.Ref(eref,identifier)
```

Python only method to get an atom reference. See **atom atom** command.

atom replace

```
atom replace ehandle label fragment/element ?fragmentlabel? ?removeh?
a.replace(replacement=,?linkatom=?,?removeh=?)
```

Replace an atom by a fragment. The fragment may be an atom, a molecule, or even a multi-molecule ensemble. The fragment parameter is either an ensemble handle, a **SMILES** string, or a list of an ensemble handle and a molecule label, identifying one molecule within that ensemble. For Python, it may also be a molecule reference. Ensembles or molecules identified by handles will not be destroyed, because the command works on a duplicate.

If the fragment parameter cannot be decoded as any of those fragment definition styles, an attempt is made to interpret is as an element or pseudo-element symbol. Deuterium and tritium isotopes may be specified as D and T, and standard query atom and superatom specifications are also understood with a syntax identical to the **atom create** command. For standard element modifications, such as by a molecule editor, the **atom change** variant of this command is preferred, because that command does not attempt to decode the fragment parameter in the other styles first and thus avoids problems with element symbols that are at the same time valid SMILES strings.

The first atom in the atom list of the selected fragment structure (which does not necessarily correspond to the lowest label in that structure) is the default link atom on the fragment. The link atom is the fragment atom which replaces the original atom in the input ensemble. A different link atom can be selected by providing a valid label (not an index) of a fragment atom as optional parameter.

All valence bonds (B_TYPE *normal*), ionic (B_TYPE *ionic*) and complex bonds (B_TYPE *complex*) to the original atom are preserved with their bond order, as are standard bond attributes (property B_FLAGS). It is possible to replace atoms with more than one neighbor, or with multiple or aromatic bonds. In the **atom change** variant of this command, it is an error if the fragment link atom cannot provide sufficient electrons to satisfy the VB bonds of the replaced atom. In **atom replace** mode, existing bonds that cannot be recreated are silently ignored. If the *removeh* flag is set, the program will attempt to find required valence electrons by removing hydrogen atoms from the link atom. If no more hydrogens can be found, electron pairs are used as a last resort, but without trying to adjust formal charges. The 2D coordinates of the link atom (property A_XY), if present, are set to the old coordinates of the replaced atom. Other properties are lost or adapted according to the merge functions of the underlying property definitions.

The return value of this command is the label (for **Tcl**) or reference (for **Python**) of the fragment link atom, which is the same as the label of the replaced atom. All atom and bond labels in the base fragment are guaranteed to be preserved, with the exception of the labels of the bonds around the replaced atom. The labels of the added fragment are generally changed, but are copied to properties A_LABEL%, B_LABEL% etc. before the merge.

Examples:

```
set ehandle [ens create CCBr]
set newlabel [atom replace $ehandle 3 [ens create Cl 0]]
set newlabel [atom change $ehandle 3 Cl]
atom replace $ehandle $newlabel {FC(F)F} 2 1
```

The second line replaces atom 3 (bromine) with a chlorine atom. The chlorine ensemble was generated without hydrogens, to Cl has a bonding electron. Also, the default link atom of the fragment is the only atom, so there cannot be any question about the fragment link location. The new label of the Cl atom is stored - but this is not really required, since it is always 3, the label of the replaced atom. Line three is the same exchange expressed in the more efficient syntax of the **atom change** command variant.

The fourth example code line replaces the chlorine atom with a CF_3 group. That group is set up by an in-line SMILES string. The fragment link atom is set to 2 (the carbon atom - labels from SMILES decoding follow the atom order). Since this fragment was generated with an extra hydrogen atom on carbon, the final parameter makes sure that this atom is removed before the replacement operation, yielding an electron on the CF_3 carbon atom for bonding to the main structure. If the hydrogen removal flag is not set, the operation will fail with this fragment. Without automatic hydrogen removal, the fragment needs to be written as `F[C](F)F` for successful replacement, with explicitly suppressed hydrogen addition at the carbon atom.

atom rings

```
atom rings ehandle label ?filterset? ?filtermode?
a.rings(?filters=?,?mode=?)
```

Standard cross-referencing command to obtain the labels or references of the rings the atom is a member of. This is explained in more detail in the section about object cross-references.

Examples:

```
atom rings $ehandle 1
atom rings $ehandle 1 [list heterocycle aroring]
```

The first example returns the labels of all rings the atom is a member of. If the atom is not in any ring, an empty list is returned. Only labels of rings in the SSSR or ESSSR ring set are returned, even if the currently computed ring set is larger. The second example filters the rings - only heteroaromatic rings are reported.

atom ringsystems

```
atom ringsystems ehandle label ?filterset? ?filtermode?
a.ringsystems(?filters=?,?mode=?)
```

Standard cross-referencing command to obtain the labels or references of the ring systems the atom is a member of. This is explained in more detail in the section about object cross-references.

Examples:

```
atom ringsystems $ehandle 1
atom ringsystems $ehandle 1 [list heterocycle aroring]
```

The first example returns the labels of the ring system the atom is a member of. If the atom is not in any ring, an empty list is returned. The second example filters the ring systems - a ring system label is obtained only if that ring system contains one or more hetero aromats.

atom set

```
atom set ehandle label ?property value?...
a.set(?property,value?,...)
a.set({property:value,...})
a.property = value
a[property] = value
```

Standard data manipulation command. It is explained in more detail in the section about setting property data.

Example:

```
atom set $ehandle 1 A_COLOR “blue”
```

The direct change of critical atom type data, such as the element `A_ELEMENT`, element symbol `A_SYMBOL`, or atom type `A_TYPE` should be avoided. It is safer to create a new atom, delete the old atom, and establish new bonds if an atom needs to be changed in its type, or to use the **`atom replace`** command. The dedicated creation, deletion and replacement commands will automatically take care of bookkeeping tasks such as electron counting for valence bonds. Also, direct setting of the element data will render most structure information invalid, since most properties depend directly or indirectly on the element composition. Careful manual locking and updating of property data is required if direct element manipulation is attempted.

The command returns the first data value.

atom show

```
atom show ehandle label propertylist ?filterset? ?parameterdict?
a.show(property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **`atom get`** command. The difference between **`atom get`** and **`atom show`** is that the latter does not attempt computation of property data, but raises an error if the data is not present and valid. For data already present, **`atom get`** and **`atom show`** are equivalent.

atom sigmas

```
atom sigmas ehandle label ?filterset? ?filtermode?
a.sigmas(?filters=?,?mode=?)
```

Standard cross-referencing command to obtain the labels or references of the σ systems the atom is a member of. This is explained in more detail in the section about object cross-references.

Examples:

```
atom sigmas $ehandle 1
```

σ systems are a rather exotic feature and not commonly used. These are essentially descriptions of bonding interactions which use s orbitals, such as normal, covalent single bonds, or the central bond in multiple bonds. A simple double bond is described with one σ system and one π system in this representation.

atom sqldget

```
atom sqldget ehandle label propertylist ?filterset? ?parameterdict?
a.sqldget(property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the *atom get* command. The differences between **`atom get`** and **`atom sqldget`** are that the latter does not attempt computation of property data, but initializes the property value to the default and returns that default, if the data is not present and valid; and that the **SQL** command variant formats the data as **SQL** values rather than for **Tcl** or **Python** script processing.

atom sqlget

```
atom sqlget ehandle label propertylist ?filterset? ?parameterdict?
a.sqlget(property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **`atom get`** command. The difference between **`atom get`** and **`atom sqlget`** is that the **SQL** command variant formats the data as **SQL** values rather than for **TCL** or **PYTHON** script processing.

atom sqlnew

```
atom sqlnew ehandle label propertylist ?filterset? ?parameterdict?
a.sqlnew(property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **`atom get`** command. The differences between **`atom get`** and **`atom sqlnew`** are that the latter forces re-computation of the property data, and that the **SQL** command variant formats the data as **SQL** values rather than for **TCL** or **PYTHON** script processing.

atom sqlshow

```
atom sqlshow ehandle label propertylist ?filterset? ?parameterdict?
a.sqlshow(property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **`atom get`** command. The differences between **`atom get`** and **`atom sqlshow`** are that the latter does not attempt computation of property data, but raises an error if the data is not present and valid, and that the **SQL** command variant formats the data as **SQL** values rather than for **TCL** or **PYTHON** script processing.

atom stereoligands

```
atom stereoligands ehandle label
a.stereoligands()
```

Get the set of ligands which define the stereochemistry of an atom. If the atom is not stereogenic, the result is a list of four empty strings for **TCL** or **`None`** values for **PYTHON**. If the atom is the center atom of an odd allene, the list contains the substituents at either end of the allene, independently sorted by atom label in ascending order for each side. If the atom is a normal tetrahedral or square planar center, the direct ligands in sorted ascending label order are returned. If one of the ligands is an electron pair, it is returned as an an empty or **`None`** list element in last position. If the atom is a pyramidal or octahedral stereo center, the normal 4-element list is expanded to include the extra atoms.

atom subcommands

```
atom subcommands
dir(Atom)
```

Lists all subcommands of the **`atom`** command. Note that this command does not require an ensemble handle, or an atom label.

atom surfaces

```
atom surfaces ehandle label ?filterset? ?filtermode?
a.surfaces(?filters=?,?mode=?)
```

Standard cross-referencing command to obtain the labels or references of surface patches the atom is associated with. This is explained in more detail in the section about object cross-references.

Example:

```
atom surfaces $ehandle $label
```

Note that individual surface patches are not required to be associated with any atom.

atom topodistance

```
atom topodistance ehandle label targetlabel ?bondclasses?
a.topodistance(target=,?bondclasses=?)
```

Compute the minimum topological distance to the second atom. By default, only VB bonds are traversed, but the optional argument allows the specification of a custom bit set of traversed bond classes.

If there is no path between the atoms, the result is -1. The distance of an atom to itself is zero.

In case many topological distances within a structure are needed, it is faster to compute property `A_TOPO_DISTANCE` once.

atom torsion

```
atom torsion ehandle label label2 label3 label4 ?property?
a.torsion(atom2=,atom3=,atom4=,?coordinateproperty=?)
```

Compute the torsion angle in degrees between the four atoms, which are considered to be linked in the specified sequence, i.e. the first three atoms defined the first plane, and the last three atoms the second plane. The torsion angle is the vector angle of the normals of the two implicitly defined planes and is always in the +/-180 degrees range.

The source property for atomic coordinates is by default `A_XYZ`, but another property can be set, which also needs to be an atomic float vector. All atoms used in a statement must be different, and possess 3D coordinates initially, or after an automatically initiated attempt to compute 3D atomic coordinate property. No bonds need to exist between the atoms.

atom uncharge

```
atom uncharge ehandle label
a.uncharge()
```

Perform a chemistry-smart transformation of the atom to remove or at least minimize its formal charge. Depending on the atom charge, type and element, this involves addition or removal of hydrogen atoms, or, if these are not available or the element has no clear hydrogen count/formal charge rules, direct editing of the formal charge and modification of the free electron count.

The command returns the number of change operations on the atom. These are either hydrogen additions and deletions, or direct formal charge changes.

atom valencecheck

```
atom valencecheck ehandle label ?nitrogenmode?
a.valencecheck(?nitrogenmode=?)
```

Perform a valence check on the atom, comparing the current bonding situation at the atom to the list of element-specific valence states in the system element table. This command is intentionally quite picky, discouraging for example the use of pentavalent nitrogen by default. For the calculation of valence, only bonds of type *normal* are taken into account. *Complex* bonds and pseudo bond types thus do not interfere in the calculation. Some more exotic metals with many different valence states, or few well-defined covalent compounds, such as *vanadium* or *rhodium*, always pass.

The handling of nitrogen in pentavalent or ionic form can be controlled by setting the optional *nitrogenmode* argument, or modifying the global **::cactvs(nitrogen_valence_check)** variable.Possible values are *xionic*, *ionic* (the default), *asis*, *pentavalent* and *xpentavalent*. These are the same values as with the **ens nitrostyle** command - please refer to that command for more information. In *asis* mode, both ionic and pentavalent forms pass.

The return value of this command is 0 for failure, 1 for pass.

Note that this command assumes that all hydrogen atoms are in place. Processing structures with Implicit hydrogen atoms is not supported.

Example:

```
atom valencecheck [ens create {CN(=O)=O}] 2
atom valencecheck [ens create {C[N+](=O)[O-]}] 2
```

These sample commands check the valence state of atom 2, the nitrogen atom in two different encodings of nitromethane. The first encoding returns 0, the second 1.

atom valcheck is a short alias.

atom vicinity

```
atom vicinity ehandle label maxdistance ?mindistance? ?filterset? ?property?
a.vicinity(maxdistance=,?mindistance=?,?filters=?,?coordinateproperty=?)
```

Get a list of the labels of atoms located in a 3D distance range from a query atom. The distance is computed from the atomic coordinates in property `A_XYZ`, or another float-vector atomic property explicitly specified. Query distances are specified in Angstrom, or whatever the default unit of a custom property is. If no minimum distance is given, it is assumed to be zero. Nevertheless, the query atom itself is never part of the returned set.

The reported atoms do not need to be bonded to the query atom directly or indirectly. If no atoms are found in the distance range, or none pass the optional filter set, an empty list results. If there are no atomic 3D coordinates, and these cannot be computed, an error is raised.

atom xdelete

```
atom xdelete ehandle ?label?...
atom xdelete ehandle all
a.xdelete()
Atom.Xdelete(eref,?alabel/aref/arefsequence?,...)
Atom.Xdelete(aref,...)
Atom.Xdelete(eref,"all")
```

This command is a variation of the **atom delete** command. The only difference is that it also deletes all hydrogen atoms the deleted atoms were bonded to.

The special atom label *all* requests deletion of all atoms. Usually, this is equivalent to **ens clear**.

The return value of the command is the number of deleted atoms.

Example:

```
atom xdelete $ehandle 1
```

This command is one of few atom subcommands which do not require an atom label. If no label is given, the command does nothing. This is useful for list expansions where the list might be empty:

```
atom xdelete $ehandle {*}$delatomlist
```

atom xhdelete

```
atom xhdelete ehandle ?label?...
atom xhdelete ehandle all
a.xhdelete()
Atom.Xhdelete(eref,?alabel/aref/arefsequence?,...)
Atom.Xhdelete(aref,...)
Atom.Xhdelete(eref,"all")
```

This command is a variation of the **`atom delete`** command. The difference is that it also deletes all hydrogen atoms the deleted atoms were bonded to, and the cut bond valences to atoms which are not hydrogen, and not deleted by the same statement, are saturated with added hydrogen atoms.

The special atom label *all* requests deletion of all atoms. Usually, this is equivalent to **`ens clear`**.

The return value of the command is the number of deleted atoms.

The bioitem Command

Bioitems are node components of biologics objects and similar to atoms, which are the node components of structure ensembles.

Bioitems are minor objects. Their associated properties start with an I_ prefix.

bioitem append

```
bioitem append bhandle label ?property value?...
i.append({?property:value,?...})
i.append(?property,value,?...)
```

Standard data manipulation command for appending property data. It is explained in more detail in the section about setting property data.

The command returns the first data value.

bioitem defined

```
bioitem defined bhandle label property
i.defined(property)
```

This command checks whether a property is defined for the bioitem. This is explained in more detail in the section about property validity checking. Note that this is *not* a check for the presence of property data! The **`biologics valid`** command is used for this purpose.

The command returns a boolean status value.

bioitem delete

```
bioitem delete bhandle ?label?...
bioitem delete bhandle all
i.delete()
Bioitem.Delete(eref,?ilabel/iref/irefsequence?,...)
Biooitem.Delete(iref,...)
Bioitem.Delete(bref,"all")
```

Delete zero or more bioitems. All biolinks which the bioitems participate in are also deleted.

The special bioitem label *all* requests deletion of all bioitems.

The return value of the command is the number of deleted bioitems.

Example:

```
bioitem delete $bhandle 1
```

This command is one of few bioitem subcommands which do not require an bioitem label. If no label is given, the command does nothing. This is useful for list expansions where the list might be empty:

```
eval bioitem delete $bhandle $delitemlist
bioitem delete $bhandle {*}$delitemlist
```

bioitem dget

```
bioitem dget bhandle label propertylist ?filterset? ?parameterdict?
i.dget(property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **`bioitem get`** command. The difference between **`bioitem get`** and **`bioitem dget`** is that the latter does not attempt computation of property data, but rather initializes the property values to the default and return that default if the data is not yet available. For data already present, **`biooitem get`** and **`bioitem dget`** are equivalent.

bioitem exists

```
bioitem exists bhandle label ?filterlist?
i.exists(?filters=?)
Bioitem.Exists(bref,label,?filters=?)
```

Check whether this bioitem exists. Optionally, a filter list can be supplied to check for the presence of specific features. The command returns 0 if the bioitem does not exist, or fails the filter, and 1 in case of successful testing.

Example:

```
bioitem exists $bhandle 99
```

bioitem fill

```
bioitem fill bhandle label ?property value?...
i.fill({property:value,...})
i.fill(?property,value?,...)
```

Standard data manipulation command for setting data, ignoring possible mismatches between the lengths of the lists of objects associated with the property and the value list. It is explained in more detail in the section about setting property data.

bioitem filter

```
bioitem filter bhandle label filterlist
i.filter(filters)
```

Check whether a bioitem passes a filter list. The return value is boolean 1 for success and 0 for failure.

bioitem get

```
bioitem get bhandle label propertylist ?filterset? ?parameterdict?
i.get(property=,?filters=?,?parameters=?)
i[property]
i.property
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For the use of the optional property parameter list argument, refer to the documentation of the **`ens get`** command.

Variants of the **`bioitem get`** command are **`bioitem new, bioitem dget, bioitem nget, bioitem show, bioitem sqldget`**, **`bioitem sqlget, bioitem sqlnew`** and **`bioitem sqlshow.`**

bioitem index

```
bioitem index bhandle label
i.index()
```

Get the index of the bioitem. The index is the position in the bioitem list of the biologics object. The first position is index 0.

bioitem jget

```
bioitem jget bhandle label propertylist ?filterset? ?parameterdict?
i.jget(property=,?filters=?,?parameters=?)
```

This is a variant of **bioitem get** which returns the result data as a **JSON** formatted string instead of **Tcl** or **Python** interpreter objects.

bioitem jnew

```
bioitem jnew bhandle label propertylist ?filterset? ?parameterdict?
i.jnew(property=,?filters=?,?parameters=?)
```

This is a variant of **bioitem new** which returns the result data as a **JSON** formatted string instead of **Tcl** or **Python** interpreter objects.

bioitem jshow

```
bioitem jshow bhandle label propertylist ?filterset? ?parameterdict?
i.jshow(property=,?filters=?,?parameters=?)
```

This is a variant of **bioitem show** which returns the result data as a **JSON** formatted string instead of **Tcl** or **Python** interpreter objects.

bioitem item

```
bioitem item bhandle label
Bioitem.Ref(bref,identifier)
```

Return the bioitem label stored in property `I_LABEL` (**Tcl**), or a minor object reference (**Python**). This is useful in case the label used in the command is not a straightforward numerical label or reference but some other item identification format.

bioitem bioitem is an alias.

bioitem links

```
bioitem links bhandle label ?filterset? ?filtermode?
i.links(?filters=?,?mode=?)
```

Standard cross-referencing command to obtain the labels or references of the biolinks which connect this bioitem fragment. This is explained in more detail in the section about object cross-references.

bioitem biolinks is a command alias.

bioitem new

```
bioitem new bhandle label propertylist ?filterset? ?parameterdict?
i.new(property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **`bioitem get`** command. The difference between **`bioitem get`** and **`bioitem new`** is that the latter forces the re-computation of the property data, regardless whether it is present and valid, or not.

bioitem nget

```
bioitem nget bhandle label propertylist ?filterset? ?parameterdict?
i.nget(property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **`bioitem get`** command. The difference between **`bioitem get`** and **`bioitem nget`** is that the latter always returns numeric data, even if symbolic names for the values are available.

bioitem ref

```
Bioitem.Ref(bref,identifier)
```

PYTHON only method to get a bioitem reference. See **`bioitem item`** command.

bioitem set

```
bioitem set bhandle label ?property value?...
i.set(?property,value?,...)
i.set({property:value,...})
i.property = value
i[property] = value
```

Standard data manipulation command. It is explained in more detail in the section about setting property data.

bioitem show

```
bioitem show bhandle label propertylist ?filterset? ?parameterdict?
i.show(property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **`bioitem get`** command. The difference between **`bioitem get`** and **`bioitem show`** is that the latter does not attempt computation of property data, but raises an error if the data is not present and valid. For data already present, **`bioitem get`** and **`bioitem show`** are equivalent.

bioitem sqldget

```
bioitem sqldget bhandle label propertylist ?filterset? ?parameterdict?
i.sqldget(property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the *atom get* command. The differences between **`bioitem get`** and **`bioitem sqldget`** are that the latter does not attempt computation of property data, but initializes the property value to the default and returns that default, if the data is not present and valid; and that the **SQL** command variant formats the data as **SQL** values rather than for **TCL** or **PYTHON** script processing.

bioitem sqlget

```
bioitem sqlget bhandle label propertylist ?filterset? ?parameterdict?
i.sqlget(property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **`bioitem get`** command. The difference between **`bioitem get`** and **`bioitem sqlget`** is that the **SQL** command variant formats the data as **SQL** values rather than for **TCL** or **PYTHON** script processing.

bioitem sqlnew

```
bioitem sqlnew bhandle label propertylist ?filterset? ?parameterdict?
i.sqlnew(property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **`bioitem get`** command. The differences between **`bioitem get`** and **`bioitem sqlnew`** are that the latter forces re-computation of the property data, and that the **SQL** command variant formats the data as **SQL** values rather than for **TCL** or **PYTHON** script processing.

bioitem sqlshow

```
bioitem sqlshow bhandle label propertylist ?filterset? ?parameterdict?
i.sqlshow(property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **`bioitem get`** command. The differences between **`bioitem get`** and **`bioitem sqlshow`** are that the latter does not attempt computation of property data, but raises an error if the data is not present and valid, and that the **SQL** command variant formats the data as **SQL** values rather than for **TCL** or **PYTHON** script processing.

bioitem subcommands

```
bioitem subcommands
dir(Bioitem)
```

Lists all subcommands of the **`bioitem`** command. Note that this command does not require a biologics handle, or a bioitem label.

The biolink Command

Biolinks connect bioitems in biologics objects. Their role is similar to bonds connecting atoms in structure ensembles.

Biolinks are minor objects. Their associated properties use a J_ prefix (mnemonical hint: these are bioitem joins).

biolink append

```
biolink append bhandle label ?property value?...
j.append({?property:value,?...})
j.append(?property,value,?...)
```

Standard data manipulation command for appending property data. It is explained in more detail in the section about setting property data.

The command returns the first data value.

biolink defined

```
biolink defined bhandle label property
j.defined(property)
```

This command checks whether a property is defined for the biolink. This is explained in more detail in the section about property validity checking. Note that this is *not* a check for the presence of property data! The **`biologics valid`** command is used for this purpose.

The command returns a boolean status value.

biolink delete

```
biolink delete bhandle ?label?...
biolink delete bhandle all
j.delete()
Biolink.Delete(bref,?jlabel/jref/jrefsequence?,...)
Biolink.Delete(jref,...)
Biolink.Delete(bref,"all")
```

Delete zero or more biolinks. The bioitems the links connect remain intact.

The special biolink label *all* requests deletion of all links.

The return value of the command is the number of deleted biolinks.

Example:

```
biolink delete $bhandle 1
```

This command is one of few biolink subcommands which do not require an biolink label. If no label is given, the command does nothing. This is useful for list expansions where the list might be empty:

```
eval biolink delete $bhandle $dellinklist
biolink delete $bhandle {*}$dellinklist
```

biolink dget

```
biolink dget bhandle label propertylist ?filterset? ?parameterdict?
j.dget(property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **`biolink get`** command. The difference between **`biolink get`** and **`biolink dget`** is that the latter does not attempt computation of property data, but rather initializes the property values to the default and return that default if the data is not yet available. For data already present, **`biolink get`** and **`biolink dget`** are equivalent.

biolink exists

```
biolink exists bhandle label ?filterlist?
b.exists(?filters=?)
Biolink.Exists(bref,label,?filters=?)
```

Check whether this biolink exists. Optionally, a filter list can be supplied to check for the presence of specific features. The command returns 0 if the biolink does not exist, or fails the filter, and 1 in case of successful testing.

Example:

```
biolink exists $bhandle 99
```

biolink fill

```
biolink fill bhandle label ?property value?...
j.fill({property:value,...})
j.fill(?property,value?,...)
```

Standard data manipulation command for setting data, ignoring possible mismatches between the lengths of the lists of objects associated with the property and the value list. It is explained in more detail in the section about setting property data.

biolink filter

```
biolink filter bhandle label filterlist
j.filter(filters)
```

Check whether a biolink passes a filter list. The return value is boolean 1 for success and 0 for failure.

biolink get

```
biolink get bhandle label propertylist ?filterset? ?parameterdict?
j.get(property=,?filters=?,?parameters=?)
j[property]
j.property
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For the use of the optional property parameter list argument, refer to the documentation of the **`ens get`** command.

Variants of the **`biolink get`** command are **`biolink new`**, **`biolink dget`**, **`biolink nget`**, **`biolink show`**, **`biolink sqldget`**, **`biolink sqlget`**, **`biolink sqlnew`** and **`biolink sqlshow`**.

biolink index

`biolink index` *`bhandle label`*
`j.index()`

Get the index of the bioitem. The index is the position in the biolink list of the biologics object. The first position is index 0.

biolink items

`biolink items` *`bhandle label ?filterset? ?filtermode?`*
`j.items(?filters=?,?mode=?)`

Standard cross-referencing command to obtain the labels or references of the bioitems which are connected by the biolink. This is explained in more detail in the section about object cross-references.

`biolink bioitems` is a command alias.

biolink jget

`biolink jget` *`bhandle label propertylist ?filterset? ?parameterdict?`*
`j.jget(property=,?filters=?,?parameters=?)`

This is a variant of **`biolink get`** which returns the result data as a **JSON** formatted string instead of **TCL** or **PYTHON** interpreter objects.

biolink jnew

`biolink jnew` *`bhandle label propertylist ?filterset? ?parameterdict?`*
`j.jnew(property=,?filters=?,?parameters=?)`

This is a variant of **`biolink new`** which returns the result data as a **JSON** formatted string instead of **TCL** or **PYTHON** interpreter objects.

biolink jshow

`biolink jshow` *`bhandle label propertylist ?filterset? ?parameterdict?`*
`j.jshow(property=,?filters=?,?parameters=?)`

This is a variant of **`biolink show`** which returns the result data as a **JSON** formatted string instead of **TCL** or **PYTHON** interpreter objects.

biolink link

`biolink link` *`bhandle label`*
`Biolink.Ref(bref,identifier)`

Return the biolink label stored in property `J_LABEL` (**TCL**), or a minor object reference (**PYTHON**). This is useful in case the label used in the command is not a straightforward numerical label or reference but some other item identification format.

`biolink biolink` is an alias.

biolink new

`biolink new` *`bhandle label propertylist ?filterset? ?parameterdict?`*
`j.new(property=,?filters=?,?parameters=?)`

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **`biolink get`** command. The difference between **`biolink get`** and **`biolink new`** is that the latter forces the re-computation of the property data, regardless whether it is present and valid, or not.

biolink nget

```
biolink nget ehandle label propertylist ?filterset? ?parameterdict?
j.nget(property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **`biolink get`** command. The difference between **`biolink get`** and **`biolink nget`** is that the latter always returns numeric data, even if symbolic names for the values are available.

biolink ref

```
Biolink.Ref(bref,identifier)
```

PYTHON only method to get a biolink reference. See **`biolink link`** command.

biolink set

```
biolink set bhandle label ?property value?...
j.set(?property,value?,...)
j.set({property:value,...})
j.property = value
j[property] = value
```

Standard data manipulation command. It is explained in more detail in the section about setting property data.

biolink show

```
biolink show bhandle label propertylist ?filterset? ?parameterdict?
j.show(property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **`biolink get`** command. The difference between **`biolink get`** and **`biolink show`** is that the latter does not attempt computation of property data, but raises an error if the data is not present and valid. For data already present, **`biolink get`** and **`biolink show`** are equivalent.

biolink sqldget

```
biolink sqldget bhandle label propertylist ?filterset? ?parameterdict?
j.sqldget(property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **`biolink get`** command. The differences between **`biolink get`** and **`biolink sqldget`** are that the latter does not attempt computation of property data, but initializes the property value to the default and returns that default, if the data is not present and valid; and that the **SQL** command variant formats the data as **SQL** values rather than for **TCL** or **PYTHON** script processing.

biolink sqlget

```
biolink sqlget bhandle label propertylist ?filterset? ?parameterdict?
j.sqlget(property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **`biolink get`** command. The difference between **`biolink get`** and **`biolink sqlget`** is that the **SQL** command variant formats the data as **SQL** values rather than for **TCL** or **PYTHON** script processing.

biolink sqlnew

```
biolink sqlnew bhandle label propertylist ?filterset? ?parameterdict?
j.sqlnew(property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **`biolink get`** command. The differences between **`biolink get`** and **`biolink sqlnew`** are that the latter forces re-computation of the property data, and that the **SQL** command variant formats the data as **SQL** values rather than for **TCL** or **PYTHON** script processing.

biolink sqlshow

```
biolink sqlshow bhandle label propertylist ?filterset? ?parameterdict?
j.sqlshow(property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **`biolink get`** command. The differences between **`biolink get`** and **`biolink sqlshow`** are that the latter does not attempt computation of property data, but raises an error if the data is not present and valid, and that the **SQL** command variant formats the data as **SQL** values rather than for **TCL** or **PYTHON** script processing.

biolink subcommands

```
biolink subcommands
dir(Biolink)
```

Lists all subcommands of the **`biolink`** command. Note that this command does not require a biologics handle, or a biolink label.

The biologics command

Biologics objects are a high-level representation of biological macromolecules. Internally, they are a network of items, connected by links. Items usually are superatoms, and links a more intricate type of bond which do not just link the superatom items, but additionally contain information about where and how the links connect to the superatoms in expanded form. Biologics can usually be expanded into normal structure ensembles with standard atoms and bonds.

Biologics are major objects. Associated properties start with a Q_ prefix.

biologics append

```
biologics append bhandle ?property value?...
b.append({?property:value,?...})
b.append(?property,value,?...)
```

Standard data manipulation command for appending property data. It is explained in more detail in the section about setting property data.

The command returns the first data value.

biologics assign

```
biologics assign bhandle srcproperty dstproperty
b.assign(srcproperty=,dstproperty=)
```

Assign property data to another property on the same biologics object. Both properties must be associated with the biologics object class. This process is more efficient than going through a pair of **`biologics get/biologics set`** commands, because in most cases no string or **Tcl/Python** script object representations of the property data need to be created.

Both source and destination properties may be addressed with field specifications. A data conversion path must exist between the data types of the involved properties. If any data conversion fails, the command fails. For example, it is possible to assign a string property to a numeric property - but only if all property values can be successfully converted to that numeric type. The reverse example case always succeeds, out-of-memory errors and similar global events excluded.

The original property data remains valid. The command variant **`biolgics rename`** directly exchanges the property name without any data duplication or conversion, if that is possible. In any case, the original property data is no longer present after the execution of this command variant.

If the properties are not associated with biologics (prefix Q_), the operation is performed on all bioitem nodes or biolinks if appropriate.

The command returns the object handle for **Tcl**, or object reference for **Python**.

Examples

```
biologics assign $bh I_IDENT I_NAME
```

biologics create

```
biologics create ?data?
biologics create attribute value...
Biologics(?data?)
Biologics(dict)
Biologics(attribute,value,...)
Biologics.Create(?data?)
Biologics.Create(dict)
Biologics.Create(attribute,value,...)
```

Create a new biologics object. There are three basic styles.

Biologics items can be created as empty shells if no arguments are used. Alternatively, if a single argument is used, they can be initialized by decoding a line notation. Supported notations currently include pack strings (see **`biologics pack`**), and 1- or 3-letter amino acid codes. The third option is to use an attribute dictionary - either as a single argument, or as a series or key-value pairs.

The command returns the new object handle or reference.

biologics dataset

```
biologics dataset bhandle ?filterlist?
b.dataset(?filters=?)
```

Return the dataset handle or reference of the dataset the biologics item is a member of. It the biologics item is not member of a dataset, or does not pass all of the optional filters, an empty string or **`None`** for **PYTHON** is returned.

Example:

```
biologics dataset $bhandle
```

biologics defined

```
biologics defined bhandle property
b.defined(property)
```

This command checks whether a property is defined for the biologics item. This is explained in more detail in the section about property validity checking. Note that this is *not* a check for the presence of property data! The **`biologics valid`** command is used for this purpose.

biologics delete

```
biologics delete ?bhandle/bhandlelist/all?...
b.delete()
Biologics.Delete("all")
Biologics.Delete(?bref/brefsequence/bhandle?,...)
```

Delete biologics objects and all their associated bioitems and biolinks. The special parameter *all* may be used to delete all biologics currently registered in the application. Alternatively, any number of biologics handles may be specified for specific object deletions.

The command returns the number of deleted biologics.

Example:

```
biologics delete all
biologics delete $bhandle
```

biologics dget

```
biologics dget bhandle propertylist ?filterset? ?parameterdict?
b.dget(property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **`biologics get`** command. The difference between **`biologics get`** and **`biologics dget`** is that the latter does not attempt computation of property data, but rather initializes the property values to the default and return that default if the data is not yet available. For data already present, **`biologics get`** and **`bioologics dget`** are equivalent.

biologics dup

```
biologics dup bhandle ?dataset? ?position?
b.dup(?target=?,?position=?)
```

Duplicate a biologics object with all minor objects and all attached data on the biologics object proper and its minor objects.

The duplicate biologics object is placed into the same dataset as the source, if it is a member of a dataset. Specifying an explicitly empty dataset argument (or **`None`** for **Python**) places the duplicate outside any dataset, regardless of the dataset membership of the source biologics object.

If the duplicate is moved to a dataset, it is appended to the dataset end by default. This happens also if the position parameter is explicitly specified as *end* or an empty string. Otherwise, the biologics object is inserted at the given position, starting with 0. If the requested position is larger than the current size of the dataset, the biologics item is appended.

Example:

```
biologics dup $bhandle
```

The command returns a new biologics handle or reference.

biologics exists

```
biologics exists bhandle ?filterlist?
b.exists(?filters=?)
Biologics.Exists(bref,?filters=?)
```

Check whether a biologics handle is valid. The command returns boolean 0 or 1. Optionally, the biologics item may be filtered by a standard filter list, and if it does not pass the filter, it is reported as not valid.

Example:

```
biologics exists $bhandle
```

biologics expr

```
biologics expr bhandle expression
b.expr(expression)
```

Compute a standard **SQL**-style property expression for the biologics item. This is explained in detail in the chapter on property expressions.

biologics fill

```
biologics fill bhandle ?property value?...
b.fill({?property:value,...})
b.fill(?property,value?,...)
```

Standard data manipulation command for setting data, ignoring possible mismatches between the lengths of the lists of objects associated with the property and the value list. It is explained in more detail in the section about setting property data.

biologics filter

```
biologics filter bhandle filterlist
n.filter(filters=)
```

Check whether the biologics object passes a filter list. The return value is boolean 1 for success and 0 for failure.

biologics get

```
biologics get bhandle propertylist ?filterset? ?parameterdict?
biologics get bhandle attribute
b.get(property=,?filters=?,?parameters=?)
b.get(attribute)
b[property/attribute]
b.property/attribute
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

Examples:

```
biologics get $hhandle {B_IDENT B_NAME}
```

yields the ID and name of the biologics object as a list. If the information is not available, an attempt is made to compute it. If the computation fails, an error results.

For the use of the optional property parameter list argument, refer to the documentation of the **`ens get`** command.

Variants of the **`biologics get`** command are **`biologics new, biologics dget, biologics jget, biologics jnew, biologics jshow, biologics nget, biologics show, biologics sqldget, biolgics sqlget, biologics sqlnew,`** and **`biologics sqlshow.`**

In addition to property data, a biologics object possesses a few attributes, which can be retrieved with the *get* command (but not by its related sister subcommands like **`dget`**, **`sqlget`**, etc.). Some of them are also modifiable via **`biologics set.`** These attributes are:

- *coords*
  If the toolkit was compiled with factory support, these are the coordinates of the object icon on its workbench, encoded as integer pair. This attribute can be changed.
- *deletable*
  Flag indicating whether the object can be deleted with a standard **`biolgics delete`** command. This attribute is read-only. Objects which are, for example, property data values cannot be deleted by standard means.
- *failures*
  If the property computation failure cache is active, return a list of all properties which have failed computation for this object after the last structural change. This attribute is read-only.
- *footer*
  If the toolkit was compiled with factory support, this is the footer of the object icon on a workbench. This attribute can be changed.

- *gflags*
  If the toolkit was compiled with factory support, this is the currently set object icon rendering flag collection.
- *header*
  If the toolkit was compiled with factory support, this is the header of the object icon on a workbench. This attribute can be changed.
- *hidden*
  Flag indicating whether the object is hidden. This is not the same as the *invisible* state. This attribute is intended to be used for rendering selections. This attribute can be changed.
- *incomplete*
  Boolean status flag indicating an aborted input operation during the read of the object from file, which returned the structure intact but without the complete set of associated data. An aborted input may be either be the result of an explicitly set input control flag, or by encountering property data which could not be decoded. This attribute is read-only.
- *invisible*
  Flag indicating whether the object is invisible. This is not the same as the *hidden* state. An invisible object is no longer accessible via its handle. This is usually the case for objects which are scheduled for deletion, but still have lingering pointer references. This attribute is read-only.
- *javaobject*
  If the toolkit was compiled with **JNI** support, this attribute reports the memory address of the **JNI** wrapper class instance, if it exists.
- *modcount*
  Object data modification count. This attribute is read-only.
- *mutexcount*
  The number of recursive mutex locks held for this object. Only supported on Linux.
- *pyobject*
  If the toolkit was compiled with Python support, this attribute reports the memory address of the Python wrapper class instance, if it exists. This attribute is read-only.
- *pyrefcount*
  If the toolkit was compiled with Python support, this attribute reports the reference count of the Python wrapper class instance, if it exists. This attribute is read-only.
- *refcount*
  If the **Tcl** interpreter is using native **Cactvs** objects instead of string-based major object handles and integer-based minor object labels to identify toolkit objects, this returns the number of **Tcl** object references active for this object. This attribute is read-only.
- *scoped*
  A boolean object visibility control flag. If set, and global control flag `::cactvs(object_scope)` is also set, the object is visible only in the**Tcl** interpreter which set the scope flag and thus claimed it. Object list commands executed in other interpreters omit this object, and attempts to decode its handle in other interpreters will fail. The most common use of this feature is the hiding of persistent chemistry objects in scripted property computation functions.
- *selected*
  Flag indicating whether the object is selected. This attribute can be changed.

- *tooltip*
  If the toolkit was compiled with factory support, this is the tooltip of the object icon on a workbench. This attribute can be changed.
- *uuid*
  An automatically generated **UUID** globally identifying the object. This attribute is read-only, different for every object, and not dependent on its contents.
- *x*
  If the toolkit was compiled with factory support, this is the *x* coordinate of the object icon on its workbench. This attribute can be changed.
- *y*
  If the toolkit was compiled with factory support, this is the *y* coordinate of the object icon on its workbench.This attribute can be changed.

biologics getparam

```
biologics getparam hhandle property ?key? ?default?
b.getparam(property=,?key=?,?default=?)
```

Retrieve a named computation parameter from valid property data. If the key is not present in the parameter list, an empty string is returned (`None` for **Python**). If the default argument is supplied, that value is returned in case the key is not found.

If the key parameter is omitted, a complete set of the parameters used for computation of the property value is returned in dictionary format.

This command does not attempt to compute property data. If the specified property is not present, an error results.

biologics hierarchy

```
biologics hierarchy bhandle ?filterlist? ?root?
b.hierarchy(?filters=?,?root=?)
```

Return the hierarchy handle or reference of the hierarchy the biologics object is part of. If the object is not member of a hierarchy, or does not pass all of the optional filters, an empty string or `None` for **Python** is returned. By default, the hierarchy object which directly contains the object is returned. If the *root* flag is set, the root hierarchy object is reported instead, which is the same only if the hierarchy has only a single level.

Example:

```
biologics hierarchy $ehandle
```

biologics index

```
biologics index bhandle
b.index()
```

Get the position of the biologics object in the object list of its dataset. If the object is not member of a dataset, -1 is returned.

biologics items

```
biologics items bhandle ?filterset? ?filtermode?
```

```
b.items(?filters=?,?mode=?)
```

Standard cross-referencing command to obtain the labels or references of the bioitems the biologics object contains as minor objects. This is explained in more detail in the section about object cross-references.

`biologics bioitems` is an alias for this command.

Example:

```
biologics items $bhandle
```

biologics jget

```
biologics jget bhandle propertylist ?filterset? ?parameterdict?
b.jget(property=,?filters=?,?parameters=?)
```

This is a variant of **`biologics get`** which returns the result data as a **JSON** formatted string instead of **TCL** or **PYTHON** interpreter objects. The command is usable only for property data, not attribute retrieval.

biologics jnew

```
biologics jnew bhandle propertylist ?filterset? ?parameterdict?
b.jnew(property=,?filters=?,?parameters=?)
```

This is a variant of **`biologics new`** which returns the result data as a **JSON** formatted string instead of **TCL** or **PYTHON** interpreter objects.

biologics jshow

```
biologics jshow bhandle propertylist ?filterset? ?parameterdict?
b.jshow(property=,?filters=?,?parameters=?)
```

This is a variant of **`biologics show`** which returns the result data as a **JSON** formatted string instead of **TCL** or **PYTHON** interpreter objects.

biologics links

```
biologics links bhandle ?filterset? ?filtermode?
b.links(?filters=?,?mode=?)
```

Standard cross-referencing command to obtain the labels or references of the biolinks the biologics object contains as minor objects. This is explained in more detail in the section about object cross-references.

`biologics biolinks` is an alias for this command.

Example:

```
biologics links $bhandle
```

biologics list

```
biologics list ?filterlist?
Biologics.List(?filters=?)
```

This command returns a list of the biologics object handles currently registered in the application. This list may optionally be filtered by a standard filter list.

biologics lock

```
biologics lock bhandle propertylist/biologics/bioitem/biolink/all ?compute?
b.lock(property=,?compute=?)
```

Lock property data of the biologics object, meaning that it is no longer subject to the standard data consistency manager control. The data consistency manager deletes specific property data if anything is done to the biologics object or its minor object components which would invalidate the information. Property data remains locked until is it explicitly unlocked.

The property data to lock can be selected by providing a list of the following identifiers:

- Property names
  Valid property instances on the biologics object or biologics minor objects are locked. If the boolean *compute* flag is set, an attempt is made to compute the property if it is not yet present. Otherwise, a request to lock non-existent data is silently ignored. It is not possible to lock individual property fields.
- *all*
  All valid biologics, bioitems and biolinks properties are locked. The compute flag is ignored.
- *bioitem*
  All valid bioitems properties are locked. The compute flag is ignored.
- *biolink*
  All valid biolink properties are locked. The compute flag is ignored.
- *biologics*
  All valid biologics properties are locked. The compute flag is ignored.

The lock can be released by a **`biologics unlock`** command.

The return value is the original biologics handle or reference.

biologics max

```
biologics max bhandle propertylist ?filterset?
b.max(property=,?filters=?)
```

Get the maximum values of the properties named in the *propertylist* parameter. The return value of the command is a list of the maximum property values.

While it is possible to work with biologics object properties, this is pointless since there is only a single instance of a biologics property per biologics object. Usually, bioitem or biolink minor object properties are tested. The objects whose property values are used for the determination of the maximum values may optionally be filtered by a standard filter set. If no objects pass the filter, the result is an empty list.

biologics metadata

```
biologics metadata bhandle property ?field ?value??
b.metadata(property=,?field=?,?value=?)
```

Obtain property metadata information, or set it. The handling of property metadata is explained in more detail in its own introductory section. The related commands **`biologics setparam`** and

`biologics getparam` can be used for convenient manipulation of specific keys in the computation parameter field. Metadata can only be read from or set on valid property data.

biologics min

```
biologics min bhandle propertylist ?filterset?
b.min(property=,?filters=?)
```

Get the minimum values of the properties named in the *propertylist* parameter. The return value of the command is a list of the maximum property values.

While it is possible to work with biologics object properties, this is pointless since there is only a single instance of a biologics property per biologics object. Usually, bioitem or biolink minor object properties are tested. The objects whose property values are used for the determination of the minimum values may optionally be filtered by a standard filter set. If no objects pass the filter, the result is an empty list.

biologics move

```
biologics move bhandle ?datasethandle|remotehandle? ?position?
b.move(?target=?,?position=?)
```

Make the biologics object a member of a dataset, or remove it from a dataset. If the dataset handle or reference parameter is omitted, or is an empty string, or **`None`** for **PYTHON**, the object is removed from its current dataset. The dataset handle or reference may be the name of a remote dataset for moving objects over a network connection.

If a target dataset handle or reference is specified, the object is added to the dataset, if allowed by the acceptance bits of the dataset, and removed from any dataset it was member of before the execution of the command. By default the object is added to the end of the dataset object list, but the final optional parameter allows the specification of a dataset object list index. The first position is index zero. If the parameter value *end* is used, or the index is bigger than the current number of dataset objects minus one, the object is appended as per the default. It is legal to use this command for moving objects within the same dataset.

Another special position value is *random* or *rnd*. This value moves to the object to a random position in the dataset. Using this mode with remote datasets is currently not supported.

The dataset handle cannot be a transient dataset.

The return value of the command is the dataset of the object prior to the move operation. It is either a dataset handle/reference, or an empty string (**TCL**) or **`None`** (**PYTHON**) if it was not member of a dataset.

biologics mutex

```
biologics mutex bhandle mode
b.mutex(mode)
```

Manipulate the object mutex. During the execution of a script command, the mutex of the major object(s) associated with the command are automatically locked and unlocked, so that the operation of the command is thread-safe. This applies to builds that support multi-threading, either by allowing multiple parallel script interpreters in separate threads or by supporting helper threads for the acceleration of command execution or background information processing.

This command locks major objects for a period of time that exceeds a single command. A lock on the object can only be released from the same interpreter thread that set the lock. Any other threaded interpreters, or auxiliary threads, block until a mutex release command has been executed when accessing a locked command object. This command supports the following modes:

- *lock*
  Increase the recursive mutex lock count on the object. The command returns the current lock count after the command, excluding the transient single-command lock.
- *reset*
  Release all persistent locks on the object, if they exist.
- *test*
  Return the current persistent lock count on the object. This excludes the transient per-command lock.
- *unlock*
  Decrease the recursive lock count on the object. The command returns the current lock count after the command, excluding the transient single-command lock. Unlocking an object which has not been persistently locked results in an error.

There is no *trylock* command variant because the command already needs to be able to acquire a transient object mutex lock for its execution.

The command returns the current lock count.

biologics need

```
biologics need bhandle propertylist ?mode? ?parameterdict?
b.need(property=,?mode=?,?parameters=?)
```

Standard command for the computation of property data, without immediate retrieval of results. This command is explained in more detail in the section about retrieving property data.

The return value is the original biologics object handle or reference.

biologics new

```
biologics new bhandle propertylist ?filterset? ?parameterdict?
b.new(property=,?mode=?,?parameters=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **`biologics get`** command. The difference between **`biologics get`** and **`biologics new`** is that the latter forces the re-computation of the property data, regardless whether it is present and valid, or not.

biologics nget

```
biologics nget bhandle propertylist ?filterset? ?parameterdict?
b.nget(property=,?mode=?,?parameters=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **`biologics get`** command. The difference between **`biologics get`** and **`biologics nget`** is that the latter always returns numeric data, even if symbolic names for the values are available.

biologics nnew

```
biologics nnew bhandle propertylist ?filterset? ?parameterdict?
b.nnew(property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading object data and attributes. It is explained in more detail in the section about retrieving property data.

For examples, see the **`biologics get`** command. The difference between **`biologics get`** and **`biologics nnew`** is that the latter always returns numeric data, even if symbolic names for the values are available, and that property data re-computation is enforced.

biologics pack

```
biologics pack bhandle ?maxsize? ?requestprops? ?suppressedprops?
?compressionlib?
b.pack(?maxsize=?,?requestprops=?,?suppressedprops=?,?compressionlib=?)
```

Pack the biologics object and all its bioitem and biolink components into a base64-encoded compressed serialized object string. This string does not contain any non-printable characters and is a full dump of the internal state of the object, omitting only property data that was declared to be so easily re-computed that a dump is not worthwhile. Further object relationships, such as datasets the object might be a member in are not saved.

The maximum size of the object string (default -1, meaning unlimited size) can be configured by the optional *maxsize* parameter. The size is specified in bytes. If the pack string would be longer than the maximum size, an error results.

The other optional property parameter lists allow to request a specific property set to be part of the package, even if it normally would not be included, and to explicitly omit properties from the dump. No property computation is performed, and suppressed properties are not purged from the biologics object.

Hierarchies can be restored from a packed object string by the **`biologics unpack or biologics create`** commands.

The biologics object and its minor objects remain in existence after using this command.

The default compression library is *zlib*. Other useful variants include *lzo* and *gzip* (and there are other internal types), but these may not be available on all builds due to license issues, and you need to specify the compression library when a dataset is unpacked. It is generally recommended to stay with *zlib*.

The return value of this command is the packed string.

In **Python**, biologics objects support the standard *pickle/unpickle* protocol.

Example:

```
set dbstring [biologics pack $hhandle]
```

biologics properties

```
biologics properties filehandle ?pattern? ?noempty?
b.properties(?pattern=?,?noempty=?)
```

Generate a list of the names of all properties attached to the biologics object. This includes properties of bioitem and biolink minor objects which are part of the biologics object. Optionally, the list may be filtered by a string match pattern.

If the *noempty* flag is set, only properties where at least one data element is not the property default value are output. By default, the filter pattern is an empty string, and the *noempty* flag is not set.

The command may be abbreviated to **`props`** instead of the full name **`properties`**.

biologics purge

```
biologics purge bhandle propertylist/biologics/bioitem/biolink ?emptyonly?
b.purge(?properties=?,?emptyonly=?)
```

Delete property data from the biologics object. The properties may be biologics properties (prefix `Q_`), or properties of the biologics minor objects, i.e. bioitems (prefix `I_`) and biolinks (prefix `J_`). If a property marked for deletion is not found on the associated objects, it is silently ignored.

The optional boolean flag *emptyonly* allows to restrict the deletion to those properties where all the values of a property associated with a biologics object (such as on all bioitems in a network for bioitem properties, or just the single biologics property value for biologics properties) are set to the default property value.

In addition to property names, the object class names **`biologics, bioitem`** or **`biolink`** may be used. These delete all property data of that class from the biologics item. They do not delete the objects proper, e.g. all bioitems are still present after a **`biologics purge $bh bioitem`**, though without any data that was not locked.

The return value is the original object handle or reference.

biologics ref

```
Biologics.Ref(identifier)
```

Python only method to get a biologics reference from a handle or another identifier. For biologics, other recognized identifiers are biologics references, integers encoding the numeric part of the handle string, or the **`UUID`** of the biologics object.

biologics rename

```
biologics rename bhandle srcproperty dstproperty
b.rename(srcproperty=,dstproperty=)
```

This is a variant of the **`biologics assign`** command. Please refer the command description in that paragraph.

biologics set

```
biologics set bhandle ?property value?...
b.set(property,value,...)
b.set({property:value,...})
b.property = value
```

```
b[property] = value
```

Standard data manipulation command. It is explained in more detail in the section about setting property data.

biologics setparam

```
biologics setparam bhandle property ?key value?...
biologics setparam bhandle property dictionary
b.setparam(property,?key,value?...)
b.setparam(property,dict)
```

Set or update a property computation parameter in the metadata parameter list of a valid property. This command is described in the section about retrieving property data. The current settings of the computation parameters in the property definition are not changed.

biologics show

```
biologics show bhandle propertylist ?filterset? ?parameterdict?
b.show(property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **`biologics get`** command. The difference between **`biologics get`** and **`biologics show`** is that the latter does not attempt computation of property data, but raises an error if the data is not present and valid. For data already present, **`biologics get`** and **`biologics show`** are equivalent.

biologics sqldget

```
biologics sqldget bhandle propertylist ?filterset? ?parameterdict?
b.sqldget(property=,?mode=?,?parameters=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **`biologics get`** command. The differences between **`biologics get`** and **`biologics sqldget`** are that the latter does not attempt computation of property data, but initializes the property value to the default and returns that default, if the data is not present and valid; and that the **SQL** command variant formats the data as **SQL** values rather than for **TCL** or **PYTHON** script processing.

biologics sqlget

```
biologics sqlget bhandle propertylist ?filterset? ?parameterdict?
b.sqldget(property=,?mode=?,?parameters=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **`biologics get`** command. The difference between **`biologics get`** and **`biologics sqlget`** is that the **SQL** command variant formats the data as **SQL** values rather than for **TCL** or **PYTHON** script processing.

biologics sqlnew

```
biologics sqlnew bhandle propertylist ?filterset? ?parameterdict?
b.sqlnew(property=,?mode=?,?parameters=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **`biologics get`** command. The differences between **`biologics get`** and **`biologics sqlnew`** are that the latter forces re-computation of the property data, and that the **SQL** command variant formats the data as **SQL** values rather than for **TCL** or **PYTHON** script processing.

biologics sqlshow

```
biologics sqlshow bhandle propertylist ?filterset? ?parameterdict?
b.sqlshow(property=,?mode=?,?parameters=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **`biologics get`** command. The differences between **`biologics get`** and **`biologics sqlshow`** are that the latter does not attempt computation of property data, but raises an error if the data is not present and valid, and that the **SQL** command variant formats the data as **SQL** values rather than for **TCL** or **PYTHON** script processing.

biologics subcommands

```
biologics subcommands
dir(Biologics)
```

Lists all subcommands of the **`biologics`** command. Note that this command does not require a biologics handle.

biologics transfer

```
biologics transfer bhandle propertylist ?targethandle? ?targetpropertylist?
b.transfer(properties=,?target=?,?targetproperties=?)
```

Copy property data from one biologics object to another biologics object or other major object, without going through an intermediate scripting language object representation, or dissociate property data from the network. If a property in the argument property list is not already valid on the source network, an attempt is made to compute it.

If a target object is specified, the return value is the handle or reference of the target object. The source and target object cannot be the same object. In case a property associated with biologics minor objects (bioitems and biolinks), the behavior is the same as described for ensemble minor objects in the documentation of **`ens transfer`**.

If a target property list is given, the data from the source is stored as content of a different property on the target. For this, the data types of the properties must be compatible, and the object class of the target property that of the target object. No attempt is made to convert data of mismatched types. In case of multiple properties, the source property list and the target property list are stepped through in parallel. If there is no target property list, or it is shorter than the source list, unmatched entries are stored as original property values, and this implies that the object class of the source and target objects are the same.

If no target object is specified, or it is spelled as an empty string or **Python `None`**, the visible effect of the command is the same as a simple **`biologics get`**, i.e. the result is the property data value or value list. The property data is then deleted from the source object. In case the data type of the deleted property was a major object (i.e. an ensemble, reaction, table, dataset or network), it is only unlinked from the source object, but not destroyed. This means that the object handles returned by the command can henceforth the used as independent objects. They can be deleted by a normal object deletion command, and are no longer managed by the source object.

biologics unlock

```
biologics unlock bhandle propertylist/biologics/bioitem/biolink/all
b.unlock(property=)
```

Unlock property data for the biologics item, meaning that they are again under the control of the standard data consistency manager.

The property data to unlock can be selected by providing a list of the following identifiers:

- Property names or references
  Valid property instances on the biologics or its minor objects are unlocked. Non-existent data is silently ignored. It is not possible to unlock individual property fields.
- *all*
  All valid biologics object and associated minor object properties are unlocked.
- *biologics*
  This is an object class identifier. All property data which is controlled by the biologics major object and attached to the specified object class is unlocked.
- *bioitem*
  This is an object class identifier. All property data which is controlled by the biologics major object and attached to its bioitem minor objects is unlocked.
- *biologics*
  This is an object class identifier. All property data which is controlled by the biologics major object and attached to it biolink minor objects is unlocked.

Property data locks are obtained by the **`biologics lock`** command.

The return value is the original biologics object handle or reference.

biologics unpack

```
biologics unpack packstring ?compressionlib?
Biologics.Unpack(data=,?compressionlib=?)
```

Unpack a base64-encoded serialized object string which was created by a **`biologics pack`** command. The return value of this function is the handle or reference of the newly created biologics object, which is an exact duplicate of the packed original biologics object.

Biologics objects may also be unpacked by a **`biologics create`** command.

The default compression library is *zlib*. For more options, see **`biologics pack`**.

Example:

```
set packdata [biologics pack $bhandle]
set bhandle [biologics unpack $packdata]
```

biologics valid

```
biologics valid bhandle propertylist
b.valid(property/propertysequence)
```

Returns a list of boolean values indicating whether values for the named properties are currently set for the biologics object or its minor objects. No attempt at computation is made. For **Python**, where single-item lists are syntactically not the same as a single value, the return value is a single boolean if the argument was a string or a property reference, and only a single property was decoded.

biologics verify

```
biologics verify bhandle property
b.verify(property)
```

Verify the values of the specified property on the biologics object. The property data must be valid, and a biologics object or biologics minor sub-object property. If the data can be found, it is checked against all constraints defined for the property, and, if such a function has been defined, is tested with the value verification function of the property.

If all tests are passed, the return value is boolean 1, 0 if the data could be found but fails the tests, and an error condition otherwise.

The *bond* Command

The *bond* command is the generic command used to manipulate bond. The syntax of this command follows the standard schema of *command/subcommand/majorhandle/minorlabel*.

Examples:

```
bond get $ehandle 1 B_ORDER
bond atoms $ehandle 2
```

This is the list of officially supported subcommands:

bond align3d

```
bond align3d ehandle label point1 point2 ?property?
b.align3d(point1=,point2=,?coordinateproperty=?)
```

Align the 3D atomic coordinates (by default in property `A_XYZ`) of the molecule the first specified atom of the bond is a member of in such a fashion that the first bond atom is positioned on the first point argument, and the vector to the second bond atom points into the same direction as the vector from the first to the second point argument.The bond lengths are not changed in this process.

The syntax of the point arguments is the generic vector syntax as documented in the **vec** command. If the bond is selected with an identifier different than an atom pair, the first bond atom is the atom moved to the first point argument. If the atom pair specification syntax is used, the first atom in the specification list is the anchor, which may or may not be the first bond atom.

The command fails if property `A_XYZ` (or is explicitly specified alternative) is not present on the ensemble and cannot be computed.

Example:

```
bond align3d $eh {2 1} 0 x
```

Atom 2 is moved to the origin, and the bond from atom 2 to atom 1 points in x-direction, i.e. it has a 3D coordinate triple like {0.0,0.0,1.5}, with the bond length as z component. The other atomic coordinates in the molecule are adjusted accordingly.

The command does not check for coordinate overlap with atoms in other molecules in the ensemble.

In case of special bonds, the second atom may not be in the same molecule as the first. This is legal - its coordinates are only needed to compute the axis and degree of rotation - though the second atom is then not moved by the command.

The command returns the label (for **Tcl**) or reference (for **Python**) of the bond.

The command name can be shortened to *align*.

bond append

```
bond append ehandle label ?property value?...
b.append({?property:value,?...})
b.append(?property,value,?...)
```

Standard data manipulation command for appending property data. It is explained in more detail in the section about setting property data.

The command returns the first data value.

Example:

```
bond append $ehandle 1 B_LABELCOLOR "00"
```

bond atoms

```
bond atoms ehandle label ?filterset? ?filtermode?
b.atoms(?filters=?,?mode=?)
```

Standard cross-referencing command to obtain the labels or references of the atoms which form the bond. This is explained in more detail in the section about object cross-references.

Examples:

```
bond atoms $ehandle 1
bond atoms $ehandle 1 {!carbon !hydrogen} count
```

The first example returns all labels of the atoms in bond 1. The second example will compute the number of atoms in the bond which are neither carbon nor hydrogen.

bond bond

```
bond bond ehandle label
Bond.Ref(eref,identifier)
```

Standard cross-referencing command to obtain the label (for **Tcl**) or reference (for **Python**) of the bond as stored in property `B_LABEL` from a bond label, or another bond identifier, such as an atom label pair. This is explained in more detail in the section about object cross-references.

Example:

```
bond bond $ehandle [list 1 2]
```

returns the label of the bond between atoms 1 and 2, or an empty string if the bond does not exist.

bond change

```
bond change ehandle label deltabondorder/type ?deltacharge?
b.change(deltabondorder=,?deltacharge=?)
Bond.Change(eref=,atoms=,deltabondorder=,?deltacharge=?)
```

This command changes the order of a bond. It may also be used to create bonds, or to delete bonds.

As in all bond commands, the bond may either be identified by its label or equivalent descriptor, or a set of atom identifiers. In case a new bond is made, a list or tuple of atom labels or other atom identifiers is provided as parameter instead of a single bond identifier. The distinction between atom and bond references is performed via the list length of the *label* parameter. Anything with more than one list element is interpreted as an atom-based specification. The order of atoms in an atom-based specification is arbitrary.

The parameter *deltabondorder* is usually a signed integer which defines the bond order change. If it is 0, the command does nothing, if the *deltacharge* parameter is also zero or omitted. If it is less than zero, the bond order is reduced. For VB bonds, the free electron count on the atoms (property `A_FREE_ELECTRONS`) is adjusted. If the bond is not a valence bond, or the change in bond order is larger than the existing bond order, the bond is deleted. If the change in bond order is positive, and the bond type a normal VB bond, the bond order is increased, provided that the atoms have sufficient free electrons for bonding (again property `A_FREE_ELECTRONS`). A positive change in bond order for a non-VB bond has no effect.

If a bond type is specified in string form instead of a numerical bond order, the type of the bond is changed in `B_TYPE`. If it originally was a VB bond, the bond electrons are added to the atom free electron count in `A_FREE_ELECTRONS`. If the bond type is changed to a VB bond from another type, an attempt to create a single bond is made and the electron count adjusted. If the count would become negative, the operation fails. The Python interface uses the *deltabondorder* named argument also for this bond type change operation, which is slightly misleading.

If the optional *chargedelta* parameter is used, electrons which imply formal charge (property `A_FORMAL_CHARGE`) are transferred between the atoms before the bond operation. The charge difference is applied to the first atom in the specification, and implicitly the second atom is affected inversely. If the bond was specified by a bond label instead of an atom list, the internal order of the atoms in the bond is used.

Examples:

```
bond change [ens create CC] {1 2} -1
bond change [ens create CC] {2 1} -1 1
set ehandle [ens create C=O]; bond change $ehandle {1 2} -1; ens hadd $ehandle
bond change [ens create {[H+].[OH-]}] {1 2} 1 -1
```

The first example line performs a radical dissociation between the carbon atom (atoms 1 and 2 - when decoding a **SMILES** string, the atom labels correspond to the sequence in the **SMILES** string). The bond order change is -1, which cuts the bond because it is only a single bond. Since no charge modification was specified, both atoms end up with a radical electron.

The second line shows the same process as a heterolytic dissociation. The second carbon atom is the recipient of a positive charge, because it was listed first, and the charge delta is +1. The first carbon atom receives the counter-charge and bears a formal charge of -1.

The third example performs a bond order reduction on the C=O double bond, and then saturates the molecule with hydrogen. The result is a reduction of formaldehyde to methanol.

The final example is a recombination reaction of a proton and a hydroxide anion. Because the proton cannot provide any electrons for the new bond, the first step is a formal transfer of one electron (charge -1) to this atom. Implicitly, it is removed from the other atom of the newly formed bond, which is the negatively charged oxygen atom of the hydroxyl anion. The result is a neutral water molecule.

The command returns the new or old bond label for **Tcl**, or a bond reference for **Python**. If the bond was deleted, the return value is zero for **Tcl**, or **None** for **Python**.

bond create

```
bond create ehandle label ?type/order? ?order?
Bond(eref=,atoms=atomsequence/bond,?type=type/order?,?order=?)
Bond.Create(eref=,atoms=atomsequence/bond,?type=type/order?,?order=?)
b.create(?type/order?)
```

This command creates a new bond, or changes the bond order or bond type of an existing bond. In case a new bond is made, a list of atom labels or other atom identifiers is provided as parameter instead of a single bond identifier. The distinction between atom and bond references is performed via the list length of the *label* parameter. Anything with more than one list element is interpreted as an atom-based specification. The order of atoms in an atom-based specification is arbitrary. In case a new bond is created, the atoms are entered into the bond in that order. Atom orders in existing bonds are not changed.

The default bond type is a valence bond (`B_TYPE` property value is *vb*) of bond order 1. If this type of bond is created, the bond type identifier may be omitted and a bond order directly specified as an integer. Valence bonds are electron-counted. In order to succeed, the participating atoms must provide sufficient electrons (property `A_FREE_ELECTRONS`) for the bond. Both atoms must provide the same number of electrons. Charge recombination in bond formation is not supported by this command, but can be achieved with the **`bond change`** command. The free electron counts of the bond atoms are automatically updated. The toolkit does not try to generate more free electrons by deleting hydrogen atoms bonded to the bond atoms or similar operations. If this kind of intelligence is required, it must be explicitly scripted, or the **`bond hcreate`** command used.

Setting the bond order of an existing bond to 0 deletes the bond.

Besides normal valence bonds, this command can be used to create or manipulate any other bond type which is known to the toolkit. The names of bond classes understood by this command are parsed from the enumeration value of property `B_TYPE` and may be changed at runtime.

Non-VB bonds do not involve electron counting. It is possible to change the type of a bond with this command, and in case a VB bond is changed to a non-VB bond, the electrons which were used in the VB bond are assigned to the `A_FREE_ELECTRONS` properties of the atoms. In the reverse case the command only succeeds if sufficient free electrons are present. The bond order (stored in property **`B_ORDER`**) of non-VB bonds is zero and cannot be changed with this command. If the bond type is changed, the bond label may also change. Changing the bond order of an existing bond without a type change is guaranteed to preserve the bond label.

The command can be used to directly create VB bonds with attributes. In addition to a numeric bond order, the following bond types are understood which create (or change to) a VB bond and simultaneously set bond attributes:

- *s/a*
  Create or set a query bond of type *single or aromatic.* The VB bond order is one.
- *s/d*
  Create or set a query bond of type *single or double.* The VB bond order is one.
- *d/a*
  Create or set a query bond of type *double or aro.* The VB bond order is one.
- *up*
  Create or set a single bond with a up wedge from with the tip at the first atom.
- *down*
  Create or set a single bond with a down wedge from with the tip at the first atom.
- *crossed*
  Create or set a double bond with the `B_FLAGS` attribute crossed (generally interpreted as double bond of unknown stereochemistry).
- *either*
  This is an alias for *crossed*.

- *any*
  Create or set a query bond of type *any*. The VB bond order is one.
- *dotted*
  Create or set a single bond with the `B_FLAGS` display attribute *dotted*.
- *wiggly*
  Create a single bond which marks undefined double bond stereochemistry.
- *wavy*
  An alias for *wiggly*.

It is also possible to spell out the bond order (*single*, *double*, etc.) instead of using a numerical value.

The attributes `B_FLAGS`, `B_QUERY(`**`flags`**`)` and `B_QUERY(`**`order`**`)` of bonds which are created or edited with a standard attribute-less bond order are reset.

The atom list which serves as a bond identifier or atom set for a new bond may contain more than two atoms. There are bond types like 3-center bonds and R-group alternative connection points, or pseudo bond like bond angles and torsion angles which span three four, or even more atoms.

It is not possible with this command to create bonds which involve the exact same set of atoms as an existing bond but which are of different type. It is also not possible to create bonds which include the same atom more than once.

Changing or creating a bond triggers a *bondchange* invalidation event. All minor object classes depending on an unchanged bond set (such as rings and molecules) as well as all property data on the ensemble which is directly or indirectly sensitive to changes in the bond set is invalidated if it is not explicitly locked.

The return value of this command is the label of the newly created or updated bond for **Tcl**, or a bond reference for **Python**. If the bond was deleted, the return value is zero for **Tcl**, or **`None`** for **Python**.

Examples:

```
bond create [list 1 2] 2
bond create [list #3 #5]
bond create {3 4} complex
```

The first line creates a standard valence bond with bond order 2 between the atoms with labels 1 and 2, or changes the bond order to a double bond. In case of insufficient bonding electrons, an error is raised. The second example create a single bond between atoms with index (*not* label) 3 and 5. The final example creates a bond of type *complex* between atoms 3 and 4, using an abbreviated Tcl list notation. This bond does not perform valence electron counting.

The simple **Python** bond constructor syntax has one special limitation. While it is possible to delete an existing bond via **`Bond.Create()`** with bond order zero (which then returns **`None`**), this does not work due to syntactic reasons with the simple **`Bond()`** constructor, which must return a valid bond reference, or throw an error. In any case, using the **`create`** subcommand for bond deletion is counterintuitive under normal circumstances.

bond cut

```
bond cut ehandle label ?substituentatom1? ?substituentatom2?
b.cut(?substituent=?,?substituent2=?)
```

Cut the specified bond and optionally add real or pseudo atoms to the cut site. If no substituent atoms are specified, the bond is simply cut and its electrons added to the bond atoms are free electrons. This is equivalent to a **`bond delete`** command. If substituents are specified, these are immediately linked, via a single bond, to the cut site. If the cut bond had a multiple bond order, unused electrons are added to the free electron count. If the cut bond was not a VB bond, no electrons are freed, and, depending on the type of the substituents, adding a substituent can fail.

Substituents are specified as element symbols, or pseudo element identifiers. If only a single substituent is specified, both sites receive the same substituent, otherwise the first substituent applies to the first bond atom, and the second substituent to the second bond atom. This command only supports bonds with two atoms.

The command returns the atom labels of the substituents. If no substituents are added to an atom, the return value for that atom is an empty string.

Example:

```
bond cut [ens create CC] 1 *
```

This cuts the C-C bond and adds an *open site* pseudo atom marker to both carbon atoms. The return value is the list **`{9 10}`** for the atom labels (references for Python) of the new pseudo atoms.

bond defined

```
bond defined ehandle label property
b.defined(property)
```

This command checks whether a property is defined for the bond. This is explained in more detail in the section about property validity checking. Note that this is *not* a check for the presence of property data! The **`ens valid`** command is used for this purpose.

Example:

```
bond defined $ehandle 1 B_ORDER
```

checks whether bond 1 is of a type for which bond orders are defined. The return value is a boolean status.

bond delete

```
bond delete ehandle label ?label?...
bond delete ehandle all
b.delete()
Bond.Delete(eref,"all")
Bond.Delete(bref,...)
Bond.Delete(eref,?blabel/bref/brefsequence?,...)
```

Delete one or more bonds. The atoms which participate in the bonds are not deleted, but in case the bond is a standard valence bond, their free electron count (property `A_FREE_ELECTRONS`) is updated. Molecule and ring information, and other minor object classes under the control of the ensemble major object which depend on an unchanged bond set are deleted. Any property data which depends on an unchanged bond set is also invalidated, or, if the property is set up to do so, re-computed.

If the bond which should be deleted does not exist, the request is silently ignored, as long as the bond specification is syntactically correct.

The return value of this command is the total of all bonds successfully deleted.

This command does not try to save stereochemistry by transferring wedge data etc. to an adjacent bond prior to deletion. The **`xdelete`** command variant offers this feature.

Example:

```
bond delete $ehandle [list 1 2]
```

bond delfrag

```
bond delfrag ehandle label ?reverse?
b.delfrag(?reverse=?)
```

This command deletes the bond and (if the reverse flag is not set) all atoms of the minor substituent (as defined by property `B_FRAGMENT_DIRECTION`) linked by the bond. If the reverse flag is set, the major substituent is deleted instead. The cut site remains unsubstituted, i.e. no automatic addition of hydrogen is performed.

The command fails of the bond is a ring bond, or not a valence or complex bond.

The return value is the number of deleted atoms.

Example:

```
bond delfrag [ens create CCCl] {2 3}]
```

The command cuts the bond between atoms 2 and 3 (the C-Cl bond), and then deletes the minor fragment (the Cl atom in this case, because it is the one with a smaller heavy atom count). The deleted fragment is canonic, other properties besides the fragment heavy atom count are used to break ties (see property **`B_FRAGMENT_DIRECTION`**).

The command can also be fully spelled out as **`bond deletefragment`**.

bond dget

```
bond dget ehandle label propertylist ?filterset? ?parameterdict?
b.dget(property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **`bond get`** command. The difference between **`bond get`** and **`bond dget`** is that the latter does not attempt computation of property data, but rather initializes the property values to the default and return that default if the data is not yet available. For data already present, **`bond get`** and **`bond dget`** are equivalent.

bond ens

```
b.ens()
```

PYTHON-only method to get the ensemble reference from a bond reference.

bond exists

```
bond exists ehandle label ?filterlist?
b.exists(?filters=?)
Bond.Exists(eref,label,?filters=?)
```

Check whether this bond exists. Optionally, a filter list can be supplied to check for the presence of specific features, or checking of the bond type. The command returns 0 if the bond does not exist, or fails the filter, and 1 in case of successful testing.

Examples:

```
bond exists $ehandle 99
bond exists $ehandle [list 1 2]
```

The second example checks whether a bond between atoms 1 and 2 exists. Instead of using a single label, all bond labels may be substituted by a list of the labels of their atoms.

bond expr

```
bond expr ehandle label expression
b.expr(expression)
```

Compute a standard **SQL**-style property expression for the bond. This is explained in detail in the chapter on property expressions.

bond fill

```
bond fill ehandle label ?property value?...
b.fill({property:value,...})
b.fill(?property,value?,...)
```

Standard data manipulation command for setting data, ignoring possible mismatches between the lengths of the lists of objects associated with the property and the value list. It is explained in more detail in the section about setting property data.

Example:

```
bond fill $ehandle 1 A_COLOR red
```

sets the color of the first atom bond 1 participates in to *red*.

The command returns the first fill value.

bond filter

```
bond filter ehandle label filterlist
b.filter(filters)
```

Check whether a bond passes a filter list. The return value is boolean 1 for success and 0 for failure.

Example:

```
bond filter $ehandle 1 [list carbon doublebond]
```

checks whether the bond is a double bond with one or more carbon atoms.

bond flip

```
bond flip ehandle label
b.flip()
```

This utility command manipulates data of property `B_FLAGS` and possibly `B_LABEL_STEREO` (and other bond stereo descriptors) plus `A_XY` and dependent data. If these property data are not already present, the command does nothing.

The mode of operation depends on the bond order.

For single bonds. the command inverts three bit groups in `B_FLAGS`.

- *dashed/dotted*
  If the bond is a wedge bond (any of the flags *highwedge* or *lowwedge* are set), and any of the bits *dashed* or *dotted* are set, both *dashed* and *dotted* are reset. The result for structure display is that a dashed wedge becomes a solid wedge. If the bond is a wedge bond, but neither *dashed* or *dotted* are set, both bits are set. The effect for display is that a solid wedge becomes a dashed wedge. Dashed/dotted bonds which are not wedge bonds are not affected.
- *left/right*
  If the bond has the *left* or *right* bit set (the second line of double bonds is plotted to the left or right side of a central bond line and slightly shortened, instead of drawing two equivalent bond lines slightly off left and right of the main axis), the currently set *left/right* bit is reset, and the other bit set. Bonds without set *left* or *right* bits are not affected.
- *front/back*
  If the bond has one of these flags set, it is cleared and replaced by the counterpart.

For double bonds, the command inverts all present bond stereo descriptors (`B_LABEL_STEREO`, `B_CIP_STEREO`, `B_CISTRANS_STEREO`, `B_MAP_STEREO`, `B_HASH_STEREO`) if they are set to a value indicating presence of stereochemistry. Stereo-dependent properties such as `B_STEREOINFO` and `B_STEREOGENIC` are invalidated if not locked. In addition, if 2D coordinates are valid in `A_XY` and the bond is not a ring bond, the smaller half of the structure is rotated around the bond axis in pseudo-3D fashion. This involves updating `A_XY` and the bond display flags in `B_FLAGS`, and invalidation of property data dependent on these. If the stereochemistry of a ring bond is changed, the 2D coordinates are deleted. Currently, 3D atomic coordinates are not modified.

The command is usually employed in preparation of a pseudo-3D horizontal or vertical flip of a structure drawing. The bond flags are set in such a way that after mirroring the 2D coordinates, the wedge orientation and ring interior positioning of the bonds are correct in the sense that they still describe the same stereo isomer and ring double bonds are plotted with a shortened bond inside the ring.

The command returns 1 if any bits were changed, 0 otherwise.

Example:

```
bond flip $handle 1
```

bond get

```
bond get ehandle label propertylist ?filterset? ?parameterdict?
b.get(property=,?filters=?,?parameters=?)
b[property]
b.property
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

Examples:

```
bond get $ehandle 1 {B_ISAROMATIC B_ORDER}
```

yields the aromaticity status flag and the (Kekulé) bond order of bond 1 as a list. If the information is not yet available, an attempt is made to compute it. If the computation fails, an error results.

```
bond get $ehandle 1 A_ELEMENT ringatom
```

gives the elements of all atoms in the bonds which are in a ring.

For the use of the optional property parameter list argument, refer to the documentation of the **ens get** command.

Variants of the **bond get** command are **bond new, bond dget, bond nget, bond show, bond sqldget, bond sqlget, bond sqlnew** and **bond sqlshow**.

Further examples:

```
bond get $ehandle 1 R_SIZE
bond get $ehandle 1 B_FLAGS(dashed)
```

bond groups

```
bond groups ehandle label ?filterset? ?filtermode?
b.groups(?filters=?,?mode=?)
```

Standard cross-referencing command to obtain the labels or references of the groups the bond is related to. This is explained in more detail in the section about object cross-references. A bond is considered to be related to a group if all the atoms in the bond are contained in the group.

Example:

```
bond groups $ehandle 1
```

bond hadd

```
bond hadd ehandle label ?filterset? ?flags? ?chargedelta?
b.hadd(?filters=?,?flags=?,?chargedelta=?)
```

Add a standard set of hydrogens to the atoms of the bonds. If the *filterset* parameter is specified, the atoms need to pass the filter set in order to be processed.

This command only adds missing implicit hydrogen. It does not reduce the current bond order, or split the bond.

Additional operation flags may be activated by setting the *flags* parameter to a list of flag names, or a numerical value representing the bit-ored values of the selected flags. By default, the flag set is empty, corresponding to the use of an empty string or *none* as parameter value. These flags are currently supported:

- *no2dcoords*
  Do not assign 2D coordinates to the added hydrogens, even if the rest of the atoms in the ensemble have valid 2D coordinates. In any case, 2D coordinates are never added when the ensemble does no already possess valid 2D coordinates.
- *no3dcoords*
  Do not assign 3D coordinates to the added hydrogens, even if the rest of the atoms in the ensemble have valid 3D coordinates. In any case, 3D coordinates are never added when the ensemble does no already possess valid 3D coordinates.
- *noanions*
  Do not add hydrogen to atoms with a negative formal charge.
- *noatoms*
  Do not add hydrogen to atoms without any bonds.

- *nocations*
  Do not add hydrogen to atoms with a positive formal charge.
- *noelements*
  Do not add hydrogen if the ensemble consists purely of isolated metal atoms, which probably represent the material in elementary form, or as an alloy.
- *noexcessvalences*
  Similar to *nohighvalences*, but hydrogen is not added to any atom which is not in its lowest standard bonded valence state.
- *nofixatomtext*
  Do not adjust property `A_TEXTLABEL` (if present) by removing references to implicit H from it on atoms where hydrogen is added. For example, by default "NHCOOEt" becomes "NCOOEt" after adding an instantiated hydrogen to the nitrogen atom. This reduces confusion on the hydrogen status when rendering all atoms.
- *nohighvalences*
  Do not add hydrogen to atoms which already exceed their lowest standard valence minus any formal charge. This option only applies to elements which have a defined lowest standard valence (this is configurable via the element table).
- *nomemory*
  Do not remember the added hydrogen atoms as automatically added. Normally, a flag is retained as part of the atom information which distinguishes atoms which were added by automatic processing, such as hydrogen addition, from those which were originally input.
- *nometals*
  Do not attempt to add hydrogen to atoms which are metals (as defined in the system element table).
- *nospecial*
  Do not perform hydrogen addition to atoms which participate in non-standard bonds (all bonds with `B_TYPE` not *normal*).
- *keepflags*
  For expert use only. Do not discard min/max values and property scope flags for atom properties when hydrogen is added.
- *protonate*
  Add a single proton to the atom. The charge of the atom is increased, only a single hydrogen is added regardless of the standard number of missing hydrogens, and this command *will* issue the standard property invalidation event for atom and bond changes.
- *resetmemory*
  Reset the origin flag described above for all atoms in the ensemble. All current atoms appear to be part of the original atom set.

If a charge delta parameter is specified, the atomic charge and free electrons of the atoms are adapted accordingly before the hydrogens are added. The manipulation of the charge usually changes the number of added hydrogen atoms. It is not possible to change the charge in such a way that the number of free electrons would become negative. This parameter is included for the sake of compatibility with the **`atom hadd`** command. It is rarely useful for bonds.

Adding hydrogens with this command, except if the *protonate* flag is set, is less destructive to the property data set of the ensemble than adding them with individual **`atom create/bond create`** commands, because many properties are designed to be indifferent to explicit hydrogen status changes, but are invalidated if the structure is changed in other ways.

The command returns the number of hydrogens which were added in total to both atoms.

bond hcreate

```
bond hcreate ehandle label ?type/order? ?order?
Bond.Hcreate(eref=,atoms=atomsequence/bond,?type=type/order?,?order=?)
b.hcreate(?type/order?)
```

This is a variant of the **`bond create`** command. The difference to the normal bond creation command is that this version attempts to delete hydrogen atoms on the bonded atoms if this prevents illegal free electron counts or valences.

All parameters and return values are the same as for **`bond create`**.

bond hstrip

```
bond hstrip ehandle label ?flags? ?chargedelta?
b.hstrip(?flags=?,?chargedelta=?)
```

This command removes hydrogens from the atoms of the selected by. By default, all hydrogen atoms are removed.

The *flags* parameter can be used to make the operation more selective. It may be a list of the following flags:

- *deprotonate*
  If this flag is set, a single proton is removed from the bond atoms. This command variant *does* issue a standard atom and bond change property invalidation event, and it always ends processing after removing the first proton. Proton removal decreases the charge of the atom by one.
- *keepalphawedge*
  Keep hydrogen atoms which are bonded to an atom which is at the tip of a wedgebond. This flag excludes the case where the bond to the hydrogen atom is the wedge bond - use the *keepwedge* flag to cover this case.
- *keepisotopes*
  Keep hydrogen atoms which are isotopically labeled (including enriched/depleted ^{1}H).
- *keeporiginal*
  Hydrogen atoms which were not automatically added via a hydrogen addition command are retained. Note that these commands can be run in a mode which does not leave information about automatic addition - hydrogens added this way do not survive.
- *keepprotons*
  Keep any molecules which consist only of hydrogen atoms (such as protons, hydride anions, and molecular hydrogen).
- *keepspecial*
  If this flag is set, hydrogens which are usually displayed, such as on aldehydes, wedge bonds, carbon triple bonds or hetero atoms are retained.

- *keepwedge*
  Keep hydrogens which are at the end of a wedge bond, indicating stereochemistry.
- *normalize*
  Normalize the wedge pattern for standard cases, removing excess wedges from hydrogens if the result structure is still stereochemically defined. Hydrogens which lose their wedge in this process are no longer protected by the *keepwedge* flag.
- *wedgetransfer*
  If a hydrogen atom is removed which is at the end of a wedge, the wedge information is saved by transferring the wedge (changing its up/down status if necessary) to an adjacent, surviving bond. This flag has no effects if the *keepspecial* or *keepwedge* flags are set. This flag is set by default.

If the *flags* parameter is an empty string, or *none*, it is ignored. The default flag value is *wedgetransfer* - but the default value is overridden if any flags are set!

If a charge delta parameter is specified, the charge and free electrons of the atoms are adapted accordingly before the hydrogens are added. The manipulation of the charge changes the number of added hydrogen atoms. It is not possible to change the charge in such a way that the number of free electrons would become negative. This option is mostly provided for compatibility with the **`atom hstrip`** command. It is rarely useful for bonds.

Hydrogen stripping is not as disruptive to the ensemble data content as normal atom deletion, except in case the *deprotonate* flag is set. The system assumes that this operation is done as part of some file output or visualization preparation. However, if any new data is computed after stripping, the computation functions see the stripped structure, and proceed to work on that reduced structure without knowledge that there are implicit hydrogens.

The return value of the command is the total number of hydrogens removed from all bond atoms.

bond hydrogenate

```
bond hydrogenate ehandle label ?filterset? ?changeset?
b.hydrogenate(?filters=?,?changeset=?)
```

Reduce the bond to a single bond except if excluded by the filter set.

If a change set is supplied, its interpretation is the same as in **`bond hadd`**.

The command returns the number of added hydrogens.

Example:

```
bond hydrogenate $eh 1 {!arobond !ccbond}
```

This reduces the bond to a single bond, provided the bond is not aromatic or a C-C bond.

bond index

```
bond index ehandle label
b.index()
```

Get the index of the bond. The index is the position in the bond list of the ensemble. The first position is index 0.

Example:

```
bond index $ehandle 99
```

bond jget

```
bond jget ehandle label propertylist ?filterset? ?parameterdict?
b.jget(property=,?filters=?,?parameters=?)
```

This is a variant of **bond get** which returns the result data as a **JSON** formatted string instead of **Tcl** or **Python** interpreter objects.

bond jnew

```
bond jnew ehandle label propertylist ?filterset? ?parameterdict?
b.jnew(property=,?filters=?,?parameters=?)
```

This is a variant of **bond new** which returns the result data as a **JSON** formatted string instead of **Tcl** or **Python** interpreter objects.

bond jshow

```
bond jshow ehandle label propertylist ?filterset? ?parameterdict?
b.jshow(property=,?filters=?,?parameters=?)
```

This is a variant of **bond show** which returns the result data as a **JSON** formatted string instead of **Tcl** or **Python** interpreter objects.

bond local

```
bond local ehandle label propertylist ?filterset? ?parameterdict?
b.local(property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading and recalculating object data. It is explained in more detail in the section about retrieving property data.

Example:

```
bond local $ehandle 1 B_LABEL_STEREO
```

Note that very few computation routines currently support the local re-computation of data - in most cases, this command falls back to a global re-computation.

bond match

```
bond match ehandle label ss_ehandle ?ss_label? ?matchflags? ?ignoreflags?
   ?atommatchvar? ?bondmatchvar? ?molmatchvar?
b.match(substructure=,?substructurebond=?,?matchflags=?,?ignoreflags=?,
   ?atommatchvariable=?,?bondmatchvariable=?,?molmatchvariable=?)
```

Check whether the selected bond matches a substructure. Only the first substructure bond, or the bond selected by the substructure bond label parameter, is tested. The substructure may be part of any structure ensemble, and even be in the same ensemble as the primary command bond. Both the bond atoms and the bond proper are checked.

The precise operation of the substructure match routine can be tuned by providing a standard set of match flags and feature ignore flags. The default match flag set has set bits for the *bondorder*, *atomtree* and *bondtree* comparison features, and an empty ignore set. If a flag set is specified as an empty string, the default set is used. In order to reset the flag set, an explicit *none* value must be used. The bit options of the match flag are explained in the documentation of the **match ss** command.

The command returns 1 for a successful match, 0 otherwise. If an optional atom, bond, or molecule match variable is specified, it is set to a nested list of matching substructure/structure atom, bond or molecule labels (references for **Python**). If no match can be found, the variable is set to an empty list. In case only a bond or molecule match variable is needed, an empty string can be used to skip the unused match variable argument positions

Example:

```
set ss [ens create {[F,Cl,Br,I][C,c]} smarts]
set b_is_cxbond [bond match $ehandle $label $ss {} {} {} amap]
if {$b_is_cxbond} {
   set b_xatom [lindex [lindex $amap 0] 1]
   set b_catom [lindex [lindex $amap 1] 1]
}
```

bond mols

```
bond mols ehandle label ?filterset? ?filtermode?
b.mols(?filters=?,?mode=?)
```

Standard cross-referencing command to obtain the label(s) (references in case of **Python**) of the molecule(s) the bond is a member of. This is explained in more detail in the section about object cross-references.

Examples:

```
bond mols $ehandle 1
bond mols $ehandle 1 heterocycle
```

The first example simply returns the label(s) of the molecule the bond is a part of. Note that it is possible for bonds to span more than one molecule - this is the reason why the command name is *mols*, not *mol*. If a bond spans more than one molecule, a list of the molecule labels is reported.

The second example returns the molecule label(s) if the bond is part of a molecule which contains one or more heterocycles. If the molecule(s) do not contain a heterocycle, an empty list is returned.

bond neighbors

```
bond neighbors ehandle label ?filterset? ?filtermode? ?anchoratomlabel?
b.neighbors(?filters=?,?mode=?,?anchoratom=?)
```

This command retrieves neighbor atoms of the bond. The atoms which participate in the bond are not included.

By default, a list with the labels (references in case of **Python**) of the atoms passing the optional filter set is the result. The retrieval mode may optionally be changed by supplying a filter mode specification list as in the standard cross referencing commands, such as *count* or *exclude*. Both parameters may be set to an empty list or entirely omitted if the default function is needed.

This command supports a special *filtermode* parameter in addition to the standard set (*exists*, *count*, *exclude*, *include*). The *bonds* parameter, followed by a bit set combination from the allowed values *ring*, *sidechain* or *bridge* can be used for topological filtering of the traversable bonds. By default, no topological bond filtering is applied.

If the optional anchor atom label (or other atom specification) is provided, only atoms which are bonded via a VB or complex bond (`B_TYPE` *vb* or *complex*) to this atom are listed. If the anchor atom is one of the bond atoms, the effects is similar to using the **`atom neighbors`** command, except that the other bond atom(s) are excluded. If the anchor atom is not part of the bond, other neighborhood relationships can be explored.-

The command may also be invoked with the aliases **bond neighbours** and **bond ligands**.

Examples:

```
bond neighbors [ens create C=C] 1
bond neighbors [ens create c1ncccc1c1ccccc1] 6 carbon {bonds bridge count}
```

The first example returns the labels of the hydrogen atoms. The second example returns 1, because the bond has one neighbor atom which is bonded via a bridge bond.

bond new

```
bond new ehandle label propertylist ?filterset? ?parameterdict?
b.new(property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **bond get** command. The difference between **bond get** and **bond new** is that the latter forces the re-computation of the property data, regardless whether it is present and valid, or not.

bond nget

```
bond nget ehandle label propertylist ?filterset? ?parameterdict?
b.nget(property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **bond get** command. The difference between **bond get** and **bond nget** is that the latter always returns numeric data, even if symbolic names for the values are available.

bond partner

```
bond partner ehandle bondlabel atomlabel
b.partner(aref)
```

Return the label (reference in case of **Python**) of the other atom in the indicated bond. In case the bond contains more than two atoms, the first atom which is not the specified atom is returned. Using an atom label which is not participating in the bond results in an error.

Example:

```
set a2 [bond partner $ehandle $b $a1]
```

bond partners

```
bond partners ehandle bondlabel atomlabel
b.partners(aref)
```

Return the label (reference in case of **Python**) of the other atom in the indicated bond. In case the bond contains more than two atoms, a list of the atoms which are not the specified atom is returned. Using an atom label which is not participating in the bond results in an error.

bond pis

```
bond pis ehandle label ?filterset? ?filtermode?
b.pis(?filters=?,?mode=?)
```

Standard cross-referencing command to obtain the labels or references of the π systems the bond is related to. This is explained in more detail in the section about object cross-references. A bond is considered to be related to a π system if all atoms of the bond are contained in the π system.

Examples:

```
bond pis $ehandle 1
```

Get the labels (references in case of **Python**) of the π systems the bond is related to. π systems are a rather exotic feature and not commonly used. These are essentially descriptions of bonding interactions which use p or d orbitals, such as in standard covalent multiple bonds. A simple double bond is described with one σ system and one π system in this representation.

bond permute

```
bond permute ehandle label ?targetbondorder?
b.permute(?targetbondorder=?)
```

Change the bond order by rotating the bond orders of Kekulé-style alternating single/double bond aromatic rings the bond is a member of. This is a useful function if aromatic systems need to be manipulated in reaction transforms and similar circumstances. If no target bond order is specified, an attempt is made to flip the bond between single and double. If a target bond order is set, the function does nothing if the current bond has already the desired bond order. The operation does not change atomic charges and does not succeed if any valence violation is encountered. Sydnones and other exotic aromatic systems can thus fail. In case the bond is a member of more than one eligible ring, the ring which is modified should be considered arbitrary.

The function returns one if the operation succeeds (which includes doing nothing is the target bond order is already present) and zero otherwise. An error is raised only if there are problems with the arguments.

bond purge

```
bond purge ehandle label propertylist/stereo/isotope/query
b.purge(propertylist/stereo/isotope/query)
```

Reset existing property data on a bond. In case the argument is a list of property names, the value on that bond only is reset to the default value of the property. In case the property is not present on the ensemble, the command is ignored. The reset via a property list does *not* trigger a property dependency update. If that is desired, an **`ens taint`** command must be explicitly scripted.In case a reset property is an atom property instead of a bond property, the reset is executed for all bonds of the atom. Other property object class mismatches are currently not supported.

In addition to standard properties, several special pseudo property names are recognized.

The *stereo* code resets all bond-centered stereo information on the bond, and will trigger a stereo change event on the ensemble which may invalidate additional data.

The *isotope* code resets property `A_ISOTOPE` on the bond atoms, marks the isotope data as tainted and runs a data dependency check.

The *query* code resets property `B_QUERY`, marks the query data as tainted and runs a data dependency check.

The command returns the label (for **Tcl**) or reference (for **Python**) of the bond.

bond ref

```
Bond.Ref(eref,identifier)
```

Python only method to get a bond reference. See **`bond bond`** command.

bond rings

```
bond rings ehandle label ?filterset? ?filtermode?
b.rings(?filters=?,?mode=?)
```

Standard cross-referencing command to obtain the labels or references of the rings the bond is contained in. This is explained in more detail in the section about object cross-references.

Examples:

```
bond rings $ehandle 1
bond rings $ehandle 1 [list heterocycle aroring]
```

The first example returns the labels of all rings the bond is contained in. If the bond is not in any ring, an empty list is returned. Only labels of rings in the SSSR or ESSSR ring set are returned, even if the currently computed ring set is larger. The second example filters the rings - only heteroaromatic rings are reported.

bond ringsystem

```
bond ringsystem ehandle label ?filterset? ?filtermode?
b.ringsystem(?filters=?,?mode=?)
```

Standard cross-referencing command to obtain the label (reference in case of **Python**) of the ring system the bonds is contained in. This is explained in more detail in the section about object cross-references.

Examples:

```
bond ringsystem $ehandle 1
bond ringsystem $ehandle 1 [list heterocycle aroring]
```

The first example returns the labels of the ring system the bond is contained in. If the bond is not in any ring system, an empty list is returned. The second example filters the ring systems - a ring system label is obtained only if that ring system contains one or more hetero aromats.

bond rotate

```
bond rotate ehandle label angle ?property?
b.rotate(angle=,?coordinateproperty=?)
```

This command rotates one half of a molecule in 3D in property `A_XYZ` or another specified property around the axis defined by the bond. The rotation angle is specified in degrees.

The section of the molecule which is rotated is not arbitrary and independent of the order of the atoms in the bond or bond specification. If there is a difference between the centrality of the atoms of the bond, the part which is less central is rotated. Molecules other than the one containing the rotation bond are not affected. The static section of the molecule also retains all atomic coordinates.

If any rotation is performed, (which excludes cases where the rotated bond is terminal - this is a no-op), both the *3dop* and *3dglobalop* property invalidation events are generated.

The command fails if no 3D coordinates are present or can be computed, or if the bond is a ring bond.

Example:

```
bond rotate $ehandle [list 5 4] 15
```

rotates one half of a molecule around the bond between atoms 4 and 5 by 15 degrees.

The command returns the label (for **Tcl**) or reference (for **Python**) of the bond.

bond set

```
bond set ehandle label ?property value?...
b.set(?property,value?,...)
b.set({property:value,...})
b.property = value
b[property] = value
```

Standard data manipulation command. It is explained in more detail in the section about setting property data.

Example:

```
bond set $ehandle 1 B_COLOR "blue"
```

The direct change of critical bond type data, such as the bond order `B_ORDER`, or bond type `B_TYPE` should be avoided. Instead, the bond manipulation commands **`bond create`** and **`bond change`** should be used. The dedicated creation, deletion and modification commands automatically take care of bookkeeping tasks such as electron counting for valence bonds. Also, direct setting of the bond data renders most structure information invalid, since most properties depend directly or indirectly on the bond type and order. Careful manual locking and updating of property data is required if direct bond manipulation is attempted.

The command returns the first data value.

bond show

```
bond show ehandle label propertylist ?filterset? ?parameterdict?
b.show(property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **`bond get`** command. The difference between **`bond get`** and **`bond show`** is that the latter does not attempt computation of property data, but raises an error if the data is not present and valid. For data already present, **`bond get`** and **`bond show`** are equivalent.

bond sigmas

```
bond sigmas ehandle label ?filterset? ?filtermode?
b.sigmas(?filters=?,?mode=?)
```

Standard cross-referencing command to obtain the labels or references of the σ systems the bond is participating in. This is explained in more detail in the section about object cross-references. A bond is considered to be related to a σ system if all atoms of the bond are contained in the σ system.

Examples:

```
bond sigmas $ehandle 1
```

σ systems are a rather exotic feature and not commonly used. These are essentially descriptions of bonding interactions which use s orbitals, such as normal, covalent single bonds, or the central bond in multiple bonds. A simple double bond is described with one σ system and one π system in this representation.

bond sqldget

```
bond sqldget ehandle label propertylist ?filterset? ?parameterdict?
b.sqlget(?filters=?,?mode=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **`bond get`** command. The differences between **`bond get`** and **`bond sqldget`** are that the latter does not attempt computation of property data, but initializes the property value to the default and return that default, if the data is not present and valid; and that the **SQL** command variant formats the data as **SQL** values rather than for **Tcl** or **Python** script processing.

bond sqlget

```
bond sqlget ehandle label propertylist ?filterset? ?parameterdict?
b.sqlget(?filters=?,?mode=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **`bond get`** command. The difference between **`bond get`** and **`bond sqlget`** is that the **SQL** command variant formats the data as **SQL** values rather than for **Tcl** or **Python** script processing.

bond sqlnew

```
bond sqlnew ehandle label propertylist ?filterset? ?parameterdict?
b.sqlnew(?filters=?,?mode=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **`bond get`** command. The differences between **`bond get`** and **`bond sqlnew`** are that the latter forces re-computation of the property data, and that the **SQL** command variant formats the data as **SQL** values rather than for **Tcl** or **Python** script processing.

bond sqlshow

```
bond sqlshow ehandle label propertylist ?filterset? ?parameterdict?
b.sqlshow(?filters=?,?mode=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **`bond get`** command. The differences between **`bond get`** and **`bond sqlshow`** are that the latter does not attempt computation of property data, but raises an error if the data is not present and valid, and that the **SQL** command variant formats the data as **SQL** values rather than for **Tcl** or **Python** script processing.

bond stereoligands

```
bond stereoligands ehandle label
b.stereoligands()
```

Return a list of the bond ligand atoms which define the stereochemistry of the bond. The length of the list is always four elements. If the bond is not stereogenic, four empty strings for **Tcl** and four **None** values for **Python** are returned. If the bond is the center bond of an even allene (including cases where rings substitute for an allene double bond), the allene endpoint ligands are returned, each side independently sorted by atom labels in ascending order. If the bond is a normal stereobond, the direct bond ligands are returned, again in independently sorted atom label order for each side. If an electron pair is part of the stereochemistry, it is included as an empty string or **None** after the partner ligand at each bond side.

bond subcommands

```
bond subcommands
dir(Bond)
```

Lists all subcommands of the **bond** command. Note that this command does not require an ensemble handle, or a bond label.

bond surfaces

```
bond surfaces ehandle label ?filterset? ?filtermode?
b.surfaces(?filters=?,?mode=?)
```

Standard cross-referencing command to obtain the labels or references of the surface patches a bond is related to. This is explained in more detail in the section about object cross-references. A bond is considered to be related to a surface element if it is linked to any of the atoms in the bond.

Example:

```
bond surfaces $ehandle 1
```

bond transform

```
bond transform ehandle label SMIRKSlist ?direction? ?reactionmode? ?selectionmode?
?flags? ?overlapmode? ?{?exclusionmode? excludesslist}? ?maxstructures? ?timeout?
?maxtransforms? ?niterations?
b.transform(transforms=,?direction=?,?reactionmode=?,?selectionmode=?,?flags=?,?
overlapmode=?,?maxstructures=?,?timeout=?,?maxtransforms=?,?iterations=?)
```

This command is complex, but nearly identical to the **ens transform** command. Please refer to that command for a full description of the command arguments.

The difference to **ens transform** is that the argument bond must be matched in the transform pattern. If the transform matches only elsewhere, no operation is performed. It is not required that the bond is actually modified during the course of the transform - it suffices if it is part of the source pattern match.

The return value is, just as with the **ens transform** command, a list of result ensembles.

bond uncharge

```
bond uncharge ehandle label
b.uncharge()
```

This command attempts to combine opposing charges on the atoms of the bond by increasing the bond order. If the bond order was increased, the result is 1, otherwise 0. The result for non-VB bonds is always 0.

Example:

```
bond uncharge [ens create {C[N+]([O-])=O}] {2 3}
```

This example converts the single bond between the nitrogen cation (atom 2) and the oxygen anion (atom 3) to a double bond and thus neutralizes the charges on the atoms.

The command returns the label (for **Tcl**) or reference (for **Python**) of the bond.

bond xdelete

```
bond xdelete ehandle label ?label?...
bond xdelete ehandle all
b.xdelete()
Bond.Xdelete(eref,?label?,...)
Bond.Xdelete(bref,...)
Bond.Xdelete(eref,"all")
```

Delete one or more bonds, while trying to maintain stereochemistry. The atoms which participate in the bonds are not deleted, but in case the bond is a standard valence bond, their free electron count (property `A_FREE_ELECTRONS`) is updated. Molecule and ring information, and other minor object classes under the control of the ensemble major object which depend on an unchanged bond set are deleted. Any property data which depends on an unchanged bond set is also invalidated, or, if the property is set up to do so, re-computed. Wedge bonds and other stereochemistry information tied to a deleted bond is transferred to an adjacent bond prior to deletion, if possible.

If the bond which should be deleted does not exist, the request is silently ignored, as long as the bond specification is syntactically correct.

The return value of this command is the total of all bonds successfully deleted.

Example:

```
bond xdelete $ehandle [list 1 2]
```

The chemobj command

This is an utility command to help with the generalization of commands applied to the various explicitly named chemistry objects in the toolkit.

The following subcommands are supported:

chemobj class

```
chemobj class objecthandle
```

Returns the class or **Tcl** command name associated with the major object identified by the handle. This command may also be invoked as **`chemobj tclcommand`**.

chemobj eval

```
chemobj eval subcommandname objecthandle ?args?...
```

Execute the **Tcl** command associated with the object. The subcommand name, object handle and optional arguments are all passed to that object-specific command in that order.

This command is intended to make it easier to exploit the regular structure of the chemistry **Tcl** commands, providing an easy method to invoke the same functionality on different kinds of major objects without the need to inspect the handles or perform other checks to identify the object type. This command invokes the original class command. It does not perform error checking on its own. It is only safe to be used with subcommands which use identical syntax, or at least an identical syntax with a specific argument set, for all object classes a script is expected to encounter.

Example:

```
chemobj eval purge $handle $proplist
```

Above command purges the properties in the list from the passed object, regardless whether the object is, for example, an ensemble, a reaction, a dataset or a table.

chemobj get

```
chemobj get class/handle attribute
```

Query information on a chemistry object class. The identifier may either be a class name, as returned by **`chemobj list`**, or, for major objects, a valid object handle. The following attributes are recognized:

- *address_city*
  The city part of the author contact address.
- *address_country*
  The country part of the author contact address, following the ISO3166 standard.
- *address_state*
  The state part of the author contact address. Empty if not applicable.
- *address_street*
  The street address part of the author contact address. Includes floor, house number, etc.
- *address_zip*
  The **ZIP** code or other applicable postal code of the author contact address.

- *affiliation*
  The institution the author works for.
- *affiliationduns*
  The **DUNS** registration ID of the affiliated institution. This is primarily useful for US government projects.
- *affiliationurl*
  The **URL** of the affiliated institution.
- *author*
  The author of the object class.
- *authorurl*
  A **URL** with information on the author, or an empty string if unset.
- *class*
  The class name of the object class, which is also the name of the associated **TCL** command. This is the same as the *tclcommand* attribute.
- *classuuid*
  The base class **UUID** of this object class.
- *comment*
  A free-form string comment on the object class.
- *date*
  The data the module source was last modified.
- *doi*
  A digital object identifier for the object class, if defined.
- *email*
  The email address of the author of the object class.
- *infourl*
  A **URL** with information on the object class, or an empty string if unset.
- *keywords*
  A list of keywords associated with the object class.
- *labelproperty*
  The name of the property which is used for set minor object labels, e.g. `A_LABEL` for the *atom* class, and an empty string for ensembles.
- *license*
  The license class associated with this object class. Setting the license to a standard type updates the associated **URL** with a standard location.
- *licenseurl*
  A **URL** with details about the object class license.
- *literature*
  A free-form literature reference.

- *name*
  The primary name of the object class.
- *orcid*
  The **ORCID** code of the author (see www.orcid.org).
- *ownerclass*
  The class name of the major object which controls objects of this class, e.g *ens* for *atom*. For major objects, the class and owner class are the same.
- *path*
  The repository path for displaying hierarchical repository trees. This attribute is independent of any file system paths.
- *phone*
  A contact phone number of the author.
- *propertyprefix*
  The standard prefix (without the underscore) for the names of properties associated with this object class.
- *references*
  Cross references of the object class. This is a nested list of class **UUID**s and reference type tags.
- *regid*
  A numerical registration ID assigned to registered object classes.
- *tclcommand*
  The same as the *class* attribute.
- *version*
  The version of the object class. This is a string in a 1.2.3 (or shortened) style
- *versionuuid*
  The **UUID** associated with this object class version.

chemobj list

`chemobj list` *`?pattern?`*

List the currently loaded chemistry object classes with their primary name.

chemobj pythoncommand

`chemobj pythoncommand` *`objecthandle`*

Returns the **Python** class associated with the major object identified by the handle. This command may also be invoked as **`chemobj pythonclass`**.

chemobj tclcommand

`chemobj tclcommand` *`objecthandle`*

Returns the class or **Tcl** command name associated with the major object identified by the handle. This command may also be invoked as **`chemobj class`**.

The *connection* Command

The connection command is used to access information about links in generic network objects (see **`network`** command). In many respects the behavior of connection objects in networks is comparable to that of bonds in ensembles, and the commands for handling connections are similarly structured. For example, just like bonds can be identified by a list of the participating atoms, connections can be selected by a list of the participating vertices.

Pseudo connection labels *first*, *last* and *random* are special values, which select the first connection in the connection list, the last, or a random connection.

The command *edge* is an alias for *connection*, allowing the use of a more standard nomenclature, but without the benefit of a matching prefix on the names of connection properties.

The following connection commands are supported:

connection append

```
connection append nhandle label ?property value?...
c.append({property:value,...})
c.append(?property,value?,...)
```

Standard data manipulation command for appending property data. It is explained in more detail in the section about setting property data.

The command returns the first data value.

Example:

```
connection append $nhandle $c C_IDENT "_v2"
```

connection connection

```
connection connection nhandle label
Connection.Ref(nref,identifier)
```

Return the connection label stored in property `C_LABEL` (TCL), or a reference (PYTHON). This is useful in case the label is not the straightforward connection label or minor object reference, but some other specification type, such as a vertex pair.

Example:

```
connection connection $nh [list $v1 $v2]
```

connection create

```
connection create nhandle vertex_list ?directionality? ?property value?...
Connection(nref,vertexsequence,?directionality=?,?property,value?...)
Connection.Create(nref,vertexsequence,?directionality=?,?property,value?...)
```

Create a new connection which links the vertices specified in the vertex list argument. In contrast to the handling of bonds in ensembles, there can be multiple connections with the same set of vertices in a network, and self-links (linked the same vertex as source and destination) are allowed. It is also possible to link more than two vertices by a connection. In some contexts the order of the vertices registered in a connection matters, i.e. the connections are interpreted as directional.

By setting the *directionality* parameter, the presence of a connection duplicate can be detected and in that case the old label (for **Tcl**) or reference (for **Python**) of the existing connection is returned, instead of creating a new connection. The possible values of the directionality parameter are *undefined* (or 0, the default, no duplicate checking), *undirected* (or 1, the vertices are matched regardless of the order in the specified list) and *directed* (or 2, the vertices are matched in the same order as in the argument list).

The magic vertex label value *new* can be used in the vertex list to automatically create one or more new vertices with this command instead of referring to existing vertices. The new vertices are added to the vertex list and have the same properties as vertices created explicitly with a **`vertex create`** command.

An initial set of property values for the new or re-used connection can be set by the optional property/value arguments.

The command returns the new or old connection label for **Tcl**, or the corresponding reference for **Python**.

connection defined

```
connection defined nhandle label property
c.defined(property)
```

This command checks whether a property is defined for the connection. This is explained in more detail in the section about property validity checking. Note that this is *not* a check for the presence of property data! The **`network valid`** command is used for this purpose

connection delete

```
connection delete nhandle ?label?...
connection delete nhandle all
c.delete()
Connection.Delete(nref,?label?,...)
Connection.Delete(cref,...)
Connection.Delete(nref,"all")
```

Delete specific or all connection from the network. The vertices participating in the deleted connections remain in the network.

The command returns the number of deleted connections.

connection dget

```
connection dget nhandle label propertylist ?filterset? ?parameterdict?
c.dget(property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the *connection get* command. The difference between **`connection get`** and **`connection dget`** is that the latter does not attempt computation of property data, but rather initializes the property values to the default and return that default if the data is not yet available. For data already present, **`connection get`** and **`connection dget`** are equivalent.

connection network

```
c.network()
```

Python-only method to get the network reference from a connection reference.

connection exists

```
connection exists nhandle label ?filterlist?
c.exists(?filters=?)
Connection.Exists(nref=,label=,?filters=?)
```

Check whether this connection exists. Optionally, a filter list can be supplied to check for the presence of specific features. The command returns boolean 0 if the connection does not exist, or fails the filter, and 1 in case of successful testing.

Examples:

```
connection exists $nhandle 99
connection exists $nhandle [list 1 2]
```

The second example checks whether a connection between vertices 1 and 2 exists. Instead of using a single label, all connection labels may be substituted by a list of the labels of their vertices.

connection filter

```
connection filter nhandle label filterlist
c.filter(filters)
```

Check whether a connection passes a filter list. The return value is boolean 1 for success and 0 for failure.

connection get

```
connection get nhandle label propertylist ?filterset? ?parameterdict?
c.get(property=,?filters=?,?parameters=?)
c[property]
c.property
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

Example:

```
connection get $nhandle [list $v1 $v2] C_ONTOLOGY_LINK
```

yields the ontology link type data of connection 1 as a list. If the information is not yet available, an attempt is made to compute it. If the computation fails, an error results.

For the use of the optional property parameter list argument, refer to the documentation of the **`ens get`** command.

Variants of the **`connection get`** command are **`connection new`**, **`connection dget`**, **`connection nget`**, **`connection show`**, **`connection sqldget`**, **`connection sqlget`**, **`connection sqlnew`** and **`connection sqlshow`**.

connection index

```
connection index nhandle label
c.index()
```

Get the index of the connection. The index is the position in the connection list of the network. The first position is index 0.

Example:

```
connection index $nhandle 99
```

connection jget

```
connection jget nhandle label propertylist ?filterset? ?parameterdict?
c.jget(property=,?filters=?,?parameters=?)
```

This is a variant of **connection get** which returns the result data as a **JSON** formatted string instead of **TCL** or **PYTHON** interpreter objects.

connection jnew

```
connection jnew nhandle label propertylist ?filterset? ?parameterdict?
c.jnew(property=,?filters=?,?parameters=?)
```

This is a variant of **connection new** which returns the result data as a **JSON** formatted string instead of **TCL** or **PYTHON** interpreter objects.

connection jshow

```
connection jshow nhandle label propertylist ?filterset? ?parameterdict?
c.jshow(property=,?filters=?,?parameters=?)
```

This is a variant of **connection show** which returns the result data as a **JSON** formatted string instead of **TCL** or **PYTHON** interpreter objects.

connection network

```
c.network()
```

PYTHON-only method to get the network reference from a connection reference.

connection new

```
connection new nhandle label propertylist ?filterset? ?parameterdict?
c.new(property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **connection get** command. The difference between **connection get** and **connection new** is that the latter forces the re-computation of the property data, regardless whether it is present and valid, or not.

connection nget

```
connection nget nhandle label propertylist ?filterset? ?parameterdict?
c.nget(property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **`connection get`** command. The difference between **`connection get`** and **`connection nget`** is that the latter always returns numeric data, even if symbolic names for the values are available.

connection ref

```
Connection.Ref(nref,identifier)
```

PYTHON only method to get a connection reference. See **`connection connection`** command.

connection set

```
connection set nhandle label ?property value?...
c.set(?property,value?,...)
c.set({property:value,...})
c.property = value
c[property] = value
```

Standard data manipulation command. It is explained in more detail in the section about setting property data.

Example:

```
connection set $nhandle 1 C_IDENT "bla"
```

connection show

```
connection show nhandle label propertylist ?filterset? ?parameterdict?
c.show(property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **`connection get`** command. The difference between **`connection get`** and **`connection show`** is that the latter does not attempt computation of property data, but raises an error if the data is not present and valid. For data already present, **`connection get`** and **`connection show`** are equivalent.

connection sqldget

```
connection sqldget nhandle label propertylist ?filterset? ?parameterdict?
c.sqldget(property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **`connection get`** command. The differences between **`connection get`** and **`connection sqldget`** are that the latter does not attempt computation of property data, but initializes the property value to the default and return that default, if the data is not present and valid; and that the **SQL** command variant formats the data as **SQL** values rather than for **TCL** or **PYTHON** script processing.

connection sqlget

```
connection sqlget nhandle label propertylist ?filterset? ?parameterdict?
c.sqlget(property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **`connection get`** command. The difference between **`connection get`** and **`connection sqlget`** is that the **SQL** command variant formats the data as **SQL** values rather than for **TCL** or **PYTHON** script processing.

connection sqlnew

```
connection sqlnew nhandle label propertylist ?filterset? ?parameterdict?
c.sqlnew(property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **`connection get`** command. The differences between **`connection get`** and **`connection sqlnew`** are that the latter forces re-computation of the property data, and that the **SQL** command variant formats the data as **SQL** values rather than for **TCL** or **PYTHON** script processing.

connection sqlshow

```
connection sqlshow nhandle label propertylist ?filterset? ?parameterdict?
c.sqlshow(property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **`connection get`** command. The differences between **`connection get`** and **`connection sqlshow`** are that the latter does not attempt computation of property data, but raises an error if the data is not present and valid, and that the **SQL** command variant formats the data as **SQL** values rather than for **TCL** or **PYTHON** script processing.

connection subcommands

```
connection subcommands
dir(Connection)
```

Lists all subcommands of the **`connection`** command. Note that this command does not require a network handle, or a connection label.

connection vertices

```
connection vertices nhandle label ?filterset? ?filtermode?
c.vertices(?filters=?,?mode=?)
```

Standard cross-referencing command to obtain the labels or references of the vertices which are participating in the connection. This is explained in more detail in the section about object cross-references.

Examples:

```
connection vertices $nhandle 1
```

The *dataset* Command

The *dataset* command is the generic command used to manipulate datasets. The syntax of this command follows the standard schema of *command/subcommand/majorhandle*. Datasets are major objects and thus do not need any minor object labels for identification.

Example:

```
dataset get $dhandle D_SIZE
```

As explained in the introductory section on datasets, a normal persistent dataset handle may be substituted as third argument of the **`dataset`** command by an arbitrary list of dataset, ensemble, reaction, table and network handles. Substitution is only allowed in that argument position, not in case where a dataset handle is part of the command arguments of another object command, and not in a different argument position in the context of a **`dataset`** command. Such an object list is transformed into a transient dataset for the duration of the command execution. After the command has completed, the elements of the transient dataset are in most cases restored to their original state with respect to dataset membership and position, except in a few documented exceptional circumstances.

As a means to access an embedded dataset object, its handle may be replaced by the handle of the parent object where this is unambiguous, e.g.

```
ens move $eh $thandle
```

moves the ensemble into the embedded dataset of the table, while

```
dataset count $thandle
```

treats the table argument as part of a transient dataset as described above.

This is the list of currently officially supported subcommands:

dataset add

```
dataset add dhandle objhandle ?position?
d.add(object=,?position=?)
d += object
```

Add an object to the dataset, relocating it from a current dataset if it exists. If no position is specified, the object is appended to the rear of the dataset object list. The position can either be a numerical zero-based index, or any string beginning with ‘e’ to indicate the end position.

If the object handle identifies a (local) dataset, and the target dataset does not accept datasets as members, all objects in the source dataset are instead moved to the new dataset, and then the source dataset is destroyed. If ensembles, reactions, tables or networks are moved, they are unlinked from any current datasets, but these original datasets themselves persist.

This dataset command is equivalent to issuing a move command from the object.

The command returns the dataset handle for **Tcl**, or the dataset reference for **Python**. The numerical operator shortcut for **Python** adds the object to the end of the dataset.

Example:

```
dataset add $dh $eh end
ens move $eh $dh end
```

These two commands are equivalent.

dataset addthread

```
dataset addthread dhandle ?body?
dataset addthread dhandle count body
dataset addthread dhandle count substitutiondict body
d.addthread(?count=?,?dictionary=?,?script=?)
```

Add one or more **Tcl** script threads to the dataset. By default, a single thread is added, but by setting the *count* parameter to a higher number multiple threads with the same script body can be added simultaneously, up to a maximum of 32 threads per dataset. It is possible to use this command to add additional threads to a dataset which already has attached threads. These older threads remain active.

The thread script code is always **Tcl** code, even if the command is issued from a **Python** interpreter. This is due to limitations in the **Python** thread model and described in more detail in the general **Python** scripting introduction.

The optional substitution dictionary contains a set of percent-prefixed keys and replacement values, following the **Tk** event procedure model. All such replacements are made before the script is passed to the thread interpreters. A single default substitution replacing the character sequence **%D** with the handle of the current dataset is always predefined and cannot be redefined. Replacement token keys (but not necessarily their values) are single case-depended characters, ignoring an optional percent prefix character. Within the script, percent signs which should be preserved as such must be doubled, just like in **Tk** event substitution commands.

The dataset threads are compatible to those of the standard **Tcl** threads package. Dataset-associated threads are automatically created in *preserved* state, and a **thread::wait** command is automatically appended at the end of the script, so they can be sent additional tasks via the **thread::send** commands. If no script body is specified, the initial script consists only of the wait command. Threads can be canceled or joined only if they are stopped the **thread::wait** statement.

When a dataset is deleted, all threads associated with this dataset need first to be joined, and this can only happen if they have finished processing the main body script and are all in their idle state in the **thread::wait** command. Object deletion is postponed until this condition is met. A global join on all currently executing dataset threads is automatically performed when the program exits, before any object clean-up tasks are run. An application where dataset threads are stuck and do not reach their **thread::wait** cancellation points cannot be cleanly exited.

Duplicating datasets does not duplicate any associated threads.

The presence of threads on a dataset has consequences for the behavior of the **dataset wait** and **dataset pop** commands, as well as object insertion commands associated with other major object classes (e.g. **ens move**, or **molfile read**). Please refer to the respective paragraphs for details. The size control mechanism of datasets in the *auto* mode is also dependent on the presence of absence of linked dataset threads.

Example:

```
dataset addthread $dh 1 [dict create %T $th] {
   while {1}
      set eh [dataset pop %D]
      if {$eh==""} break
      if {[catch {ens get $eh E_CANONIC_TAUTOMER} eh_canonic]} {
         ens delete $eh
         continue
      }
      if {[catch {ens get $eh_canonic E_DESCRIPTORS}]} {
         ens delete $eh
         continue
      }
      table addens %T $eh_canonic
      ens delete $eh
   }
}
```

This code creates a processing thread on the dataset which computes properties on newly arriving ensembles, stores the data in a table (note the table handle substitution via the replacement dictionary) and then deletes the ensemble. The **dataset pop** command returns an empty string when it is known no more data will arrive, and otherwise blocks until an object for popping is available. This is managed by setting the *eod* dataset attribute from feeder threads.

The return value of the command is a list of the **Tcl** thread IDs of the newly created threads. These are suitable for use in the **dataset jointhreads** command or any standard **Tcl** thread package command.

dataset append

```
dataset append dhandle ?property value?...
d.append({?property:value,?...})
d.append(?property,value,?...)
```

Standard data manipulation command for appending property data. It is explained in more detail in the section about setting property data.

The command returns the first data value.

Example:

```
dataset append $dhandle D_NAME "_new"
dataset append $dhandle eod 1
```

dataset assign

```
dataset assign dhandle srcproperty dstproperty
d.assign(srcproperty=,dstproperty=)
```

Assign property data to another property on the same ensemble. Both properties must be associated with the same object class. This process is more efficient than going through a pair of **dataset get/dataset set** commands, because in most cases no string or **Tcl/Python** script object representations of the property data need to be created.

Both source and destination properties may be addressed with field specifications. A data conversion path must exist between the data types of the involved properties. If any data conversion fails, the command fails. For example, it is possible to assign a string property to a numeric property - but only if all property values can be successfully converted to that numeric type. The reverse example case always succeeds, out-of-memory errors and similar global events excluded.

The original property data remains valid. The command variant **`dataset rename`** directly exchanges the property name without any data duplication or conversion, if that is possible. In any case, the original property data is no longer present after the execution of this command variant.

If the properties are not associated with datasets (prefix `D_`), the operation is performed on all dataset member objects.

The command returns the object handle for **Tcl**, or object reference for **Python**.

Example:

```
dataset assign $dhandle A_XY A_XY%
```

This code snippet creates backup atomic 2D layout coordinates on all dataset ensembles or reactions.

dataset biologics

```
dataset biologics dhandle ?filterset? ?filtermode? ?recursive?
d.tables(?filters=?,?mode=?,?recursive=?)
```

Return a list of all the handles or references of the biologics in the dataset. Other objects in the dataset (ensembles, reactions, datasets, networks) are ignored. The object list may optionally be filtered by the filter list, and the result further modified by a standard filter mode argument.

If the *recursive* flag is set, and the dataset contains other datasets as objects, biologics in these nested datasets are also listed.

Example:

```
set n [dataset biologics $dhandle {} count]
```

dataset cancelthreads

```
dataset cancelthreads ?all?
dataset cancelthreads dhandle ?all?
dataset cancelthreads dhandle threadid...
Dataset.Cancelthreads()
d.cancelthreads("all")
d.cancelthreads()
d.cancelthreads(?threadid?,...)
```

Cancel (or more precisely, wait for and join) one or more threads associated with the dataset. Dataset threads can only be canceled when they are idle, executing the implicitly added **`thread::wait`** command at the end of their script. Therefore, this command is not just used for clean-up, but also useful for ascertaining that the threads have finished their tasks. The IDs of the threads associated with a dataset can be retrieved as the *threads* dataset attribute, or saved from the return value of the original **`dataset addthread`** command. The special *all* thread ID value can be used to cancel all threads of the dataset. This can also be achieved by setting an empty thread ID parameter, or omitting it altogether. If a dataset does not possess threads, this command does nothing. If a thread marked for cancellation has not yet finished, the cancellation command is suspended until it has.

This command can also be invoked without specifying an explicit or transient dataset argument, or passing it as *all*. In that case, the thread join cleanup is run on all threads of all currently defined datasets. This function is also implicitly run when a a script exits, before performing other application cleanup operations.

Thread cancellation for all dataset threads is implicitly invoked when a dataset is deleted, so an explicit clean-up is not required. However, this also means that a dataset deletion blocks if there are still active threads. It is not possible to forcefully cancel an thread which has entered an infinite loop, so careful programming is required.

The command returns the number of canceled threads.

`dataset jointhreads` is an alias to this command.

Example:

```
dataset jointhreads $dh
dataset cancelthreads $dh [lindex [dataset get $th threads] 0]
dataset jointhreads
```

The first example waits for all threads on the specified dataset to finish. The second command waits for the completion of one specific thread, and the last command waits for all threads on all currently defined datasets.

dataset cast

```
dataset cast dhandle dataset/ens/reaction/table ?propertylist?
d.cast(objectclass=,?properties=?)
```

Transform the dataset into a different object. Depending on the target object class, the result is as follows:

- *dataset*
  Only supplied for the sake of completeness. This mode does nothing.
- *ens*
  The first ensemble contained in the dataset, or a newly created empty ensemble if no such object exists. The dataset and all its other contents are destroyed in the process.
- *reaction*
  The first reaction contained in the dataset, or a newly created empty reaction if no such object exists. The dataset and all its other contents are destroyed in the process.
- *table*
  A new table with automatically set up columns which are the union of all valid ensemble-class (E_*) and reaction-class (X_*) properties of the ensembles and reactions in the dataset, and rows with the data of these objects. In addition, these objects are moved into the internal table dataset. The input dataset, and its remaining contents which were not moved to the table, are destroyed.

If the optional property list is specified, an attempt is made to compute the listed properties before the cast operation, so that they may become a part of the new object. No error is raised if a computation fails.

The command returns the handle (reference for **Python**) of the new object, or the input object in case of mode *dataset*.

dataset clear

```
dataset clear dhandle
d.clear()
```

Delete all objects in the dataset, but keep the dataset object. The return value is the number of deleted objects.

dataset count

```
dataset count dhandle|remotehandle ?filterlist?
d.count(?filters=?)
Dataset.Count(dataset=,?filters=?)
```

Get the number of objects in the dataset. If the filter parameter is specified, only those objects which pass the filter are counted.

Example:

```
dataset count $dhandle astereogenic
```

counts the number of ensembles or reactions in the dataset with one or more potential atom stereo centers.

`dataset size` is an alias to this command.

This command can be used with remote datasets. In the case of **PYTHON**, this requires the use of the class method.

In case a simple count on a local dataset is required, without any filters, the dataset size can also be queried as attribute, as in

```
set n [dataset get $dhandle size]
```

dataset create

```
dataset create ?objecthandle/objectlist?...
Dataset(?objectref/objectsequence?,...)
Dataset.Create(?objectref/objectsequence?,...)
```

This command creates a new dataset and returns the handle of the new dataset. If the optional object handle lists are provided as arguments, the specified objects (in case of ensemble, reaction, network or table handles), or elements of the object (for a dataset handle, with default *accept* flags) are moved to the new dataset. In case the *accept* flags of the target dataset are configured to allow datasets as primary dataset objects, the source dataset argument is not implicitly replaced by its content objects but added as a single object, retaining its objects as content. Otherwise, the source dataset is emptied but remains a valid object.

Besides handles of ensembles, reactions, networks, tables, molfiles and of other datasets, which are identified with priority, any string which can be decoded in an **`ens create`** statement is also allowed as member initialization identifier.

If the **`dataset create`** statement references objects which are not usually accepted by the default settings of the *accept* dataset attribute, that attribute is automatically adjusted to allow for these objects. The accept flag modification is persistent.

Molfile objects in the object handle list are treated different from other objects. The latter are directly moved into the dataset. In the case of **`molfile`** objects, the file is read from the current position to the end (or until a termination condition configured on the **`molfile`** handle is met), and the newly *read* objects are moved into the dataset.

The command always returns the handle of the new dataset (or a reference for **Python**), never the handles of any objects which may have been placed into the dataset

Examples:

```
dataset create [list $eh1 $eh2] $dh1
```

creates a new dataset and move the two specified ensembles *$eh1* and *$eh2,* as well as everything contained in the dataset *$dh1*, into the new dataset.

```
dataset create [molfile open myfile.sdf r hydrogens add]
```

creates a dataset from the file contents, with hydrogen addition configured on the **`molfile`** handle.

```
dataset create VXPBDCBTMSKCKZ
```

Above command matches a partial InChI key, and puts all structures from the NCI resolver which matches the non-stereo/isotope-specific part of their full InChI key, into the new dataset.

```
set ::cactvs(lookupmode) „name_pattern‟
dataset create [list "+morphine +methyl"]
```

This command performs a name pattern lookup and puts all structures from the NCI resolver which contain both name fragments in one of their known names into the dataset. The name pattern string needs to be explicitly packed into a list, because otherwise it would be split into two independent list elements.

dataset dataset

```
dataset dataset dhandle ?filterlist?
d.dataset(?filters=?)
```

Get the handle (or, for **Python**, a reference) of the *container* dataset the dataset is a member of. If the dataset is not itself a dataset member, or does not pass the optional filters, an empty string is returned, or **`None`** for **Python**.

This command is *not* equivalent to **`dataset datasets`**!

dataset datasets

```
dataset datasets dhandle ?filterset? ?filtermode? ?recursive?
d.datasets(?filters=?,?mode=?,?recursive=?)
```

Return a list of all the handles or references of the datasets that are members in the dataset identified by the command argument handle. Other objects (ensembles, reactions, tables, networks) are ignored. The object list may optionally be filtered by the filter list, and the output further modified by a standard filter mode.

If the *recursive* flag is set, and the dataset contains other datasets as objects, datasets in these nested datasets are also listed.

This command is *not* equivalent of the **`dataset dataset`** command!

Example:

```
set dlist [dataset datasets $dhandle]
```

dataset defined

```
dataset defined dhandle property
d.defined(property)
```

This command checks whether a property is defined for the dataset. This is explained in more detail in the section about property validity checking. Note that this is *not* a check for the presence of property data! The **`dataset valid`** command is used for this purpose.

The command returns a boolean result.

dataset delete

```
dataset delete ?datasethandle/datasethandlelist/all?...
d.delete()
Dataset.Delete("all")
Dataset.Delete(?dref/drefsequence/dhandle?,...)
```

This command destroys datasets and everything contained therein. The special handle value *all* may be used to delete all datasets in the application at once.

The command returns the number of datasets which were successfully deleted.

Transient datasets cannot be used with this command. Neither can be datasets which are a component of another object, e.g. the internal datasets of tables or factories. These are only and automatically deleted when their parent object is destroyed. Datasets which are a property value are also undeletable by this command.

It is a common programming error to delete a dataset, or its parent object if one exists, without protecting its current member ensembles or reactions. If they are still needed in later processing they need to be explicitly transferred into another dataset or outside of it.

Examples:

```
dataset delete all
dataset move $dhandle {}; dataset delete $dhandle
```

The first example destroys all datasets defined in the current script and everything contained in them. The second example shows how to delete a dataset and preserve its contents by moving all dataset elements out prior to deletion.

dataset dget

```
dataset dget dhandle propertylist ?filterset? ?parameterdict?
d.dget(property=,?filters=?,?parameters=?)
Dataset.Dget(items,property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **`dataset get`** command. The difference between **`dataset get`** and **`dataset dget`** is that the latter does not attempt computation of property data, but rather initializes the property values to the default and return that default if the data is not yet available. For data already present, **`dataset get`** and **`dataset dget`** are equivalent.

The **Python** class method is a one-shot command. The transient dataset created from the initialization items is automatically deleted when the command finishes.

dataset dup

```
dataset dup dhandle ?targethandle? ?cleartarget?
d.dup(?target=?,?cleartarget=?)
```

If the optional arguments are not supplied, the dataset with all data attached to the dataset and all objects which are contained in it are duplicated. The command returns a new dataset handle for **Tcl**, or reference for **Python**. All duplicated objects in the new datasets also are assigned handles which can be obtained by commands such as **`dataset list $dhandle`**.

It is possible to specify a target dataset as an optional argument. In that case, no new dataset is created, and dataset-level property data on the source dataset is not copied. All objects in the source dataset are duplicated and appended to the end of the target dataset. In case the boolean target clearance flag is set, which is also the default if the parameter is omitted, the target dataset is cleared before the new objects from the source dataset are added. In this command variant, the return value of the command is the target dataset handle or reference.

Examples:

```
dataset dup $dhandle
dataset dup [list $eh1 $eh2] $dtarget 0
```

dataset ens

```
dataset ens dhandle ?filterset? ?filtermode? ?recursive?
d.ens(?filters=?,?mode=?,?recursive=?)
```

Return a list of all the handles or references of the ensembles in the dataset. Other objects (reactions, tables, datasets, networks) are ignored. The object list may optionally be filtered by the filter list, and the output further modified by a standard filter mode.

If the optional boolean *recursive* argument is set, ensembles which are a component of a reaction in the dataset are also listed. Furthermore, if the dataset contains datasets as elements, these are recursively traversed, and ensembles in these, as well as ensembles in reactions in these datasets, are listed. If the output mode of the command is a handle list, items found by recursion are appended to the result list in a straight fashion, without the creation of nested lists. By default the recursion flag is off. Regardless of the flag value, ensembles which are associated with rows of a table in the dataset, but are not themselves dataset members, are not output.

Example:

```
set elist [dataset ens $dhandle astereogenic]
```

lists those ensembles in the dataset which have one or more atoms which are potential atom stereo centers.

```
set cnt [dataset ens $dhandle {} count 1]
```

returns a count of all ensembles which are either directly members of the dataset, or indirectly as component objects of reactions in the dataset, or which are contained in datasets which are a themselves a member of the primary dataset.

dataset exists

```
dataset exists dhandle ?filterlist?
d.exists(?filters=?)
Dataset.Exists(dref,?filters=?)
```

Check whether a dataset handle or reference is valid. The command returns boolean 0 or 1. Optionally, the dataset may be filtered by a standard filter list, and if it does not pass the filter, it is reported as not valid. This command cannot be used with transient datasets.

Example:

```
dataset exists $dhandle
```

dataset expr

```
dataset expr dhandle expression
d.expr(expression)
```

Compute a standard **SQL**-style property expression for the dataset. This is explained in detail in the chapter on property expressions.

dataset extract

```
dataset extract dhandle propertylist ?filterset? ?filterprocs?
d.extract(property=,?filters=?,?filterfunctions=?)
```

This command is rather complex and closely related to the `dataset xlabel` command. It was designed for the efficient extraction of major or minor object data for filtered subsets of the dataset.

The property list parameter determines the property data which is extracted. Multiple properties may be specified, but they can only be associated with major objects and one arbitrary minor object class. So it is possible to simultaneously extract an ensemble and an atom property, but not an atom and a bond property.

The return value is a nested list of data items for every object which is encountered while traversing the dataset on the level of the minor object associated with the extraction property, or just ensembles or other major objects if no such property is selected. Every list element is itself a list which contains the extracted property values in the order they are named in the property list parameter.

The objects for which data is returned can further be filtered by a standard filter set, and additionally by a list of filter procedures (for **Tcl**, specified as procedure names) or functions (for **Python**, specified as function names or function references). These procedures or functions are called with the respective object handles/references and object labels as arguments. For example, a callback function used in an atom retrieval context would be called for each atom with its ensemble handle or reference and the atom label as arguments. If major objects without a label are checked, such as complete ensembles, 1 is passed as the label. The callback procedures are expected to return a boolean value. If it is *false* or 0, the object is not added to the returned list, and the other check procedures are no longer called.

The command currently only works on ensembles in the dataset, ignoring any reactions, tables, datasets or networks which may be present.

Because this command is primarily intended for numerical data display, the returned values are formatted as with the *nget* command, i.e. instead of enumerated values the underlying numerical values are returned.

Example:

```
set dhandle [dataset create [ens create CO] [ens create CN]]
dataset extract $dhandle [list E_NAME A_SYMBOL] !hydrogen
```

This example first creates a dataset with *methanol* and *methylamine*. The second line performs the actual extraction and returns

```
{CH4O C} {CH4O O} {CH5N C} {CH5N N}
```

This kind of extracted data is useful for the display of filtered atomic (and other minor object's) property values.

dataset filter

```
dataset filter dhandle filterset
d.filter(filters)
```

Check whether a the dataset passes a filter list. The return value is boolean 1 for success and 0 for failure. Note that only filters operating on dataset objects are applicable, not any filter for objects contained in the dataset (such as ensembles or reactions).

dataset find

```
dataset find dhandle objecthandle
d.find(objectref)
```

Get the index of the dataset object. If it cannot be found in the dataset, the result is minus one.

dataset forget

```
dataset forget dhandle ?objectclass?
d.forget(?objectclass=?)
```

This command is essentially the same as the **`ens forget`** (or **`reaction forget`**, etc) command. It is applied to all objects in the dataset.

If the object class is *dataset*, all dataset-level property data is deleted.

The command returns the dataset handle or reference, or, for **Tcl** only, an empty string if the dataset was transient.

dataset get

```
dataset get dhandle propertylist ?filterset? ?parameterdict?
dataset get dhandle attribute
d.get(property=,?filters=?,?parameters=?)
d.get(attribute)
d[property/attribute]
d.property/attribute
Dataset.Get(items,property=,?filters=?,?parameters=?)
Dataset.Get(items,attribute)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

In addition to retrieving property data, it can also be used to query dataset attributes. The set of supported attributes is detailed in the paragraph on the **`dataset set`** command.

The **Python** class method is a one-shot command. The transient dataset created from the initialization items is automatically deleted when the command finishes.

Examples:

```
dataset get $dhandle {D_NAME D_SIZE}
```

yields the name and size of the dataset as a list. If the information is not yet available, an attempt is made to compute it. If the computation fails, an error results.

```
dataset get $dhandle [list E_FORMULA E_WEIGHT]
```

gives the formula and molecular weight of all dataset ensembles. The result is delivered as a nested list. The first list are the formulas, the second list contains the weights.

Currently, it is not possible to use filters with this command (and the other retrieval command variants) which are not operating directly on the dataset object, but on objects lower in the hierarchy such as ensembles or atoms.

For the use of the optional property parameter list argument, refer to the documentation of the **`ens get`** command.

Variants of the **`dataset get`** command are **`dataset new, dataset dget, dataset jget, dataset jnew, dataset jshow, dataset nget, dataset show, dataset sqldget, dataset sqlget, dataset sqlnew,`** and **`dataset sqlshow`**.

dataset getparam

```
dataset getparam dhandle property ?key? ?default?
d.getparam(property=,?key=?,?default=?)
```

Retrieve a named computation parameter from valid property data. If the key is not present in the parameter list, an empty string is returned (**`None`** for **PYTHON**). If the default argument is supplied, that value is returned in case the key is not found.

If the key parameter is omitted, a complete set of the parameters used for computation of the property value is returned in dictionary format.

This command does not attempt to compute property data. If the specified property is not present, an error results.

Example:

```
dataset getparam $dhandle E_GIF format
```

returns the actual format of the image, which could be **GIF**, **PNG**, or various bitmap formats.

dataset hadd

```
dataset hadd dhandle ?filterset? ?flags? ?changeset?
d.hadd(?filters=?,?flags=?,?changeset=?)
```

Add a standard set of hydrogens to all ensembles and reactions in the dataset. If the *filterset* parameter is specified, only those atoms which pass the filter set are processed.

Additional operation flags may be activated by setting the *flags* parameter to a list of flag names, or a numerical value representing the bit-ored values of the selected flags. By default, the flag set is empty, corresponding to the use of an empty string or *none* as parameter value. These flags are currently supported:

- *nospecial*
  Do not perform hydrogen addition to atoms which participate in non-standard bonds (all bonds with `B_TYPE` not *normal*).
- *no2dcoords*
  Do not assign 2D coordinates to the added hydrogens, even if the rest of the atoms in the ensemble have valid 2D coordinates. In any case, 2D coordinates are never added when the ensemble does no already possess valid 2D coordinates.

- *no3dcoords*
  Do not assign 3D coordinates to the added hydrogens, even if the rest of the atoms in the ensemble have valid 3D coordinates. In any case, 3D coordinates are never added when the ensemble does no already possess valid 3D coordinates.
- *nometals*
  Do not attempt to add hydrogen to atoms which are metals (as defined in the system element table).
- *noelements*
  Do not add hydrogen if the ensemble consists purely of isolated metal atoms, which probably represent the material in elementary form, or as an alloy.
- *nomemory*
  Do not remember the added hydrogen atoms as automatically added. Normally, a flag is retained as part of the atom information which distinguishes atoms which were added by automatic processing, such as hydrogen addition, from those which were originally input.
- *resetmemory*
  Reset the origin flag described above for all atoms in the ensemble. All current atoms appear to be part of the original atom set.
- *nocations*
  Do not add hydrogen to atoms with a positive formal charge.
- *noanions*
  Do not add hydrogen to atoms with a negative formal charge.
- *nohighvalences*
  Do not add hydrogen to atoms which already exceed their lowest standard valence minus any formal charge. This option only applies to elements which have a defined lowest standard valence (this is configurable via the element table).
- *noatoms*
  Do not add hydrogen to atoms without any bonds.

Adding hydrogens with this command is less destructive to the property data set of the ensembles or reactions than adding them with individual **`atom create/bond create`** commands, because many properties are defined to be indifferent to explicit hydrogen status changes, but are invalidated if the structure is changed in other ways.

If the effects of the hydrogen addition step to the validity of the property data set should not be handled with this standard procedure, it is possible to explicitly generate additional property invalidation events by specifying a list as the optional last parameter, for example a list of *atom* and *bond* to trigger both the atom change and bond change events.

The command returns the total number of hydrogens added to all ensembles and reactions in the dataset.

Example:

```
dataset hadd $dhandle
```

dataset hdup

```
dataset hdup dhandle ?targethandle? ?cleartarget?
d.hdup(?target=?,?cleartarget=?)
```

If the optional arguments are not supplied, the dataset with all data attached to the dataset and all objects which are contained in it are duplicated with hydrogen addition. The command returns a new dataset handle for **Tcl**, or reference for **Python**. All duplicated objects in the new datasets also are assigned handles which can be obtained by commands such as **`dataset list $dhandle`**.

It is possible to specify a target dataset as an optional argument. In that case, no new dataset is created, and dataset-level property data on the source dataset is not copied. All objects in the source dataset are duplicated with hydrogen addition and appended to the end of the target dataset. In case the boolean target clearance flag is set, which is also the default if the parameter is omitted, the target dataset is cleared before the new objects from the source dataset are added. In this command variant, the return value of the command is the target dataset handle or reference.

Examples:

```
dataset dup $dhandle
dataset dup [list $eh1 $eh2] $dtarget 0
```

dataset hierarchies

```
dataset hierarchies dhandle ?filterset? ?filtermode? ?recursive?
d.tables(?filters=?,?mode=?,?recursive=?)
```

Return a list of all the handles or references of the hierarchies in the dataset. Other objects in the dataset (ensembles, reactions, datasets, networks) are ignored. The object list may optionally be filtered by the filter list, and the result further modified by a standard filter mode argument.

If the *recursive* flag is set, and the dataset contains other datasets as objects, hierarchies in these nested datasets are also listed.

This is not the same as **`dataset hierarchy`** - the latter reports the hierarchy the dataset is a member of. This command lists the hierarchies in the dataset.

Example:

```
set n [dataset hierarchies $dhandle {} count]
```

dataset hierarchy

```
dataset hierarchy dhandle ?filterlist? ?root?
d.hierarchy(?filters=?,?root=?)
```

Return the hierarchy handle or reference of the hierarchy the dataset is part of. If the dataset is not member of a hierarchy, or does not pass all of the optional filters, an empty string or **`None`** for **Python** is returned. By default, the hierarchy object which directly contains the dataset is returned. If the *root* flag is set, the root hierarchy object is reported instead, which is the same only if the hierarchy has only a single level.

This command is not the same as **`dataset hierarchies`**, which reports hierarchies in the dataset.

Example:

```
dataset hierarchy $dhandle
```

dataset hread

```
dataset hread dhandle ?datasethandle|enshandle? ?#recs|batch|all?
d.hread(?target=?,?limit=?)
```

This command provides the same functionality as **`dataset read`**, but additionally adds a stand set of hydrogen atoms to the read duplicate objects.

The command arguments are explained in the section on **`dataset read`**.

dataset hstrip

```
dataset hstrip dhandle ?flags? ?changeset?
d.hstrip(?flags=?,?changeset=?)
```

This command removes hydrogens from the dataset ensembles and reactions. By default, all hydrogen atoms in the dataset ensembles or reactions are removed.

The *flags* parameter can be used to make the operation more selective. It may be a list of the following flags:

- *keepspecial*
  If this flag is set, hydrogens which are usually displayed, such as on aldehydes, wedge bonds, carbon triple bonds or hetero atoms are retained.
- *keeporiginal*
  Hydrogen atoms which were not automatically added via a *hadd* command are retained. Note that hydrogen addition commands can be run in a mode which does not leave information about automatic addition - hydrogens added this way will also survive.
- *keepwedge*
  Keep hydrogens which are at the end of a wedge bond, indicating stereochemistry.
- *wedgetransfer*
  If a hydrogen atom is removed which is at the end of a wedge, the wedge information is saved by transferring the wedge (changing its up/down status if necessary) to an adjacent, surviving bond. This flag has no effects if the *keepspecial* or *keepwedge* flags are set. This flag is set by default.
- *keepprotons*
  Keep any molecules which consist only of hydrogen atoms (such as protons, hydride anions, and molecular hydrogen).
- *keepalphawedge*
  Keep hydrogen atoms which are bonded to an atom which is at the tip of a wedgebond. This flag excludes the case where the bond to the hydrogen atom is the wedge bond - use the *keepwedge* flag to cover this case.
- *normalize*
  Normalize the wedge pattern for standard cases, removing wedges from hydrogens if the result is still stereochemically defined. Hydrogens which lose their wedge in this process are no longer protected by the *keepwedge* flag.
- *keepisotopes*
  Keep hydrogen atoms which are isotope labels (including enriched/depleted ^1H).

If the *flags* parameter is an empty string, or *none*, it is ignored. The default flag value is *wedgetransfer* - but the default value is overridden if any flags are set!

If the *changeset* parameter is given, all property change events listed in the parameter are triggered.

Hydrogen stripping is not as disruptive to the ensemble or reaction data content as normal atom deletion. The system assumes that this operation is done as part of some file output or visualization preparation. However, if any new data is computed after stripping, the computation functions see the stripped structure, and proceed to work on that reduced structure without knowledge that there are implicit hydrogens.

Example:

```
dataset hstrip $dhandle [list keeporiginal wedgetransfer]
```

dataset index

```
dataset index dhandle
dataset index dhandle position
d.index(?position=?)
```

This command comes in two variants. The tree-word version is the generic command to check dataset membership, which is the same for all objects which can be dataset members. The second version is specific to datasets objects and retrieves object references from this dataset.

This first version gets the position of the dataset in the object list of its parent dataset. If the dataset is not part of a parent dataset, -1 is returned. This is the generic dataset membership test command variant.

This second command variant obtains the object handle or reference of the object at the specified position in this dataset. Position counting begins with zero. If the index is outside the object position range, an empty string is returned. The special value *end* may be used to address the last object. The indexed object remains in the dataset.

Note that this **`index`** command is not equivalent to the standard **`index`** command on minor objects which is used to obtain the position of the minor object in the minor object list of the controlling major object. This kind of functionality is not needed for major objects, because they are not contained in any minor object list.

Example:

```
dataset index $dhandle end
```

dataset intersect

```
dataset intersect dhandle1 dhandle2 ?property?...
d.intersect(dref2,?pref?...)
```

Perform an intersection check between two datasets. The result is a list of zero-based dataset index pairs (as in **`dataset index`**) of all identical corresponding dataset entries in both datasets, as judged by the value of the comparison property. The default comparison property is **`E_ISOTOPE_STEREO_HASH`** for full structural identity check of ensembles.

In case the first dataset contains duplicates, the index of the matching second dataset element is identical for all duplicates, and, in case the second dataset also contains corresponding duplicates, a (pseudo-)random element from among these duplicates, and the other duplicates in the second dataset are reported as not matched in the **`dataset intersect3`** command variant (see below).

The comparison property object class must match the class of the compared dataset objects (i.e. the default property is only suitable for comparison of ensembles in the datasets, but not for reactions, etc.). Objects of mismatching classes in the datasets are ignored.

Example:

```
set dh1 [dataset create CC CCC CCCC]
set dh2 [dataset create CCC CCCC CCCCC]
dataset intersect $dh1 $dh2
```

The result is **`{1 0} {2 1}`**, meaning the second (if we start counting with 1) element of the first dataset corresponds to the first element in the second, and the third element to the second.

dataset intersect3

```
dataset intersect3 dhandle1 dhandle2 ?property?...
d.intersect3(dref2,?pref?...)
```

This command is an extended variant of **`dataset intersect`**. The return value is a 3-element list comprising of a simple list of the element indices in the first dataset which are not matched, the match pair list as in dataset intersect of the equivalent elements, and a simple list containing the element indices of the second dataset which are not matched.

Example:

```
set dh1 [dataset create CC CCC CCCC]
set dh2 [dataset create CCC CCCC CCCCC]
dataset intersect3 $dh1 $dh2
```

The result is **`0 {{1 0} {2 1}} 2`**. The middle element of the result list is the same as in the example for the **`dataset intersect`** command. The first element indicates that the first (starting the count with 1) element of the first dataset was not matched, and the third element indicates that the third element of the second dataset was not matched.

dataset jget

```
dataset jget dhandle propertylist ?filterset? ?parameterdict?
d.jget(property=,?filters=?,?parameters=?)
Dataset.Jget(items,property=,?filters=?,?parameters=?)
```

This is a variant of **`dataset get`** which returns the result data as a **JSON** formatted string instead of **TCL** or **PYTHON** interpreter objects. The command is usable only for property data, not attribute retrieval.

The Python class method is a one-shot command. The transient dataset created from the initialization items is automatically deleted when the command finishes.

dataset jointhreads

```
dataset jointhreads ?all?
dataset jointhreads dhandle ?all?
dataset jointhreads dhandle threadid...
Dataset.Jointhreads()
d.jointhreads(“all”)
d.jointhreads()
d.jointhreads(?threadid?,...)
```

This is an alias for the **`dataset cancelthreads`** command. Please refer to its documentation.

dataset jnew

```
dataset jnew dhandle propertylist ?filterset? ?parameterdict?
d.jnew(property=,?filters=?,?parameters=?)
Dataset.Jnew(items,property=,?filters=?,?parameters=?)
```

This is a variant of **`dataset new`** which returns the result data as a **JSON** formatted string instead of **Tcl** or **Python** interpreter objects.

The Python class method is a one-shot command. The transient dataset created from the initialization items is automatically deleted when the command finishes.

dataset jshow

```
dataset jshow dhandle propertylist ?filterset? ?parameterdict?
d.jshow(property=,?filters=?,?parameters=?)
Dataset.Jshow(items,property=,?filters=?,?parameters=?)
```

This is a variant of **`dataset show`** which returns the result data as a **JSON** formatted string instead of **Tcl** or **Python** interpreter objects.

The Python class method is a one-shot command. The transient dataset created from the initialization items is automatically deleted when the command finishes.

dataset ldup

```
dataset ldup ?dhandlelist?...
Dataset.Ldup(?dref/drefsequence?,...)
```

Duplicate all datasets in the handle list(s) in default mode.

The return value is a single list (even if multiple source lists are used) of the duplicated ensemble handles or references. If an argument list element is an empty string (or **`None`** for **Python**), it indicates a missing object, and the output list also receives an empty string element (for **Tcl**) or **`None`** (for **Python**) at its position, without raising an error.

This command cannot be used with transient datasets.

dataset lhdup

```
dataset lhdup ?dhandlelist?...
Dataset.Lhdup(?dref/drefsequence?,...)
```

Duplicate all datasets in the handle list(s) in default mode, and add hydrogens.

The return value is a single list (even if multiple source lists are used) of the duplicated ensemble handles or references. If an argument list element is an empty string (or **`None`** for **Python**), it indicates a missing object, and the output list also receives an empty string element (for **Tcl**) or **`None`** (for **Python**) at its position, without raising an error.

This command cannot be used with transient datasets.

dataset list

```
dataset list ?dhandle?
Dataset.List(?filters=?)
d.list()
```

Without a handle argument (for **Tcl**), or called as the class method (for **Python**) the command returns a list of the handles of all existing datasets.

If (in **Tcl**) a dataset handle or transient dataset is passed as third argument, or the object method is used (for **Python**) the command returns a list of all major objects in the dataset. In the **Tcl** case, this function is different from the behavior of the *list* subcommand for other major object classes, where the optional argument is a filter list. In **Python**, the filter list variant is supported.

Examples:

```
dataset list
dataset list $dhandle
```

dataset lock

```
dataset lock dhandle propertylist/dataset/all ?compute?
d.lock(property=,?compute=?)
```

Lock property data of the dataset handle, meaning that it is no longer subject to the standard data consistency manager control. The data consistency manager deletes specific property data if anything is done to the dataset handle which would invalidate the information. Property data remains locked until is it explicitly unlocked.

The property data to lock can be selected by providing a list of the following identifiers:

- Property names (or references, in **Python**)
  Valid property instances on the file object are locked. If the boolean *compute* flag is set, an attempt is made to compute the property if it is not yet present. Otherwise, a request to lock non-existent data is silently ignored. It is not possible to lock individual property fields.
- *all*
  All valid dataset properties are locked. The compute flag is ignored.
- *dataset*
  This is an object class identifier. All property data which is controlled by the dataset major object and attached to the specified object class is locked. Since datasets do not incorporate minor objects, this identifier is equivalent to *all*.

A lock can be released by a **`dataset unlock`** command.

This command does not recurse into the objects contained in the dataset.

The return value is the dataset handle (for **Tcl**) or reference (for **Python**) or, if the dataset was transient, an empty string (for **Tcl** only).

dataset loop

```
dataset loop dhandle objvar ?maxrec? ?offset? body
d.loop(function=,?maxloop=?,?offset=?,?variable=?)
for obj in d:
```

Loop over the elements in a dataset. This command is similar to **`molfile loop`**. On each iteration, the variable is set to the handle of the current member object, and then the body code is executed. The variable refers to the original dataset element, not a duplicate. This is different from **`dataset read`**.

All operations on the current loop item are allowed, including deletion. However, the *next* object after the current item must not be deleted or moved, because it is needed for the iteration process.

If a maximum record count is set, the loop terminates after the specified number of iterations. If the maximum record argument is set to an empty string, a negative value, or *all*, the loop covers all dataset elements. This is also the default.

For **Tcl** scripts, within the loop, the standard **Tcl** `break` and `continue` commands work as expected. If the body script generates an error, the loop is exited.

If no offset is specified, the loop starts at the first element. Within the loop body, the dataset attribute *record* is continuously updated to indicate the current loop position. Its value starts with one, like file records in the `molfile loop` command.

The **Python** version of the loop method does intentionally have a different argument sequence for convenience. The function argument may either be a multi-line string (similar to the **Tcl** construct), or a function reference. Functions are called with the reference of the current loop object as single argument, and have their own context frame, so that the specification of a reference variable is not generally useful in that call style, though is is allowed. For string function blocks the code is executed in the local call frame, and the variable with the current object reference is visible locally. Script code blocks must be written with an initial indentation level of zero. Within the **Python** functions, the normal *break* and *continue* commands cannot be used to to scope limitations. Instead, the custom exceptions *BreakLoop* and *ContinueLoop* can be raised. These are automatically caught and processed in the loop body handler code.

In **Python**, there is also an object iterator so that simple loops over dataset elements can be written with a `for` statement. The dataset object iterator is of the *self* style (i.e. there is one per dataset, these are not independent objects), so nesting them is not possible on the same dataset.

Python object loop constructs and their peculiarities are discussed in more detail in the general chapter on **Python** scripting.

Example:

```
dataset loop $dh eh {
   puts „[ens get $eh E_NAME] at position[ens index $eh]"
}
```

dataset match

```
dataset match dhandle ss_ehandle ?matchflags? ?ignoreflags?
d.match(substructure=,?matchflags=?,?ignoreflags=?)
```

Perform a substructure match on all eligible objects in the dataset. The return value is the match count.

The arguments are the same as with `ens match`. The specification of variables to capture match locations is not possible in this command variant.

dataset max

```
dataset max dhandle propertylist ?filterset?
d.max(property=,?filters=?)
```

Get the maximum value of one or more properties in from the elements in the dataset. The property argument may be any property attached to dataset members, or minor objects thereof. If the *filterset* argument is specified, the maximum value is searched only for objects which pass the filter set.

Examples:

```
dataset max $dhandle E_WEIGHT
dataset max [list $ehandle1 $ehandle2] A_SIGMA_CHARGE carbon
```

The first example finds the highest molecular weight in the dataset. The second example finds the largest (most positive) Gasteiger partial charge on any carbon atom in the two argument ensembles, which form a transient dataset.

dataset metadata

```
dataset metadata dhandle property ?field ?value??
d.metadata(property=,?field=?,?value=?)
```

Obtain property metadata information, or set it. The handling of property metadata is explained in more detail in its own introductory section. The related commands **`dataset setparam`** and **`dataset getparam`** can be used for direct manipulation of specific keys in the computation parameter field. Metadata can only be read from or set on valid property data.

Valid field names are *bounds*, *comment*, *info*, *flags*, *parameters* and *unit*.

Examples:

```
array set gifparams [dataset metadata $dhandle D_GIF parameters]
dataset metadata $dhandle D_QUALITY comment "This value looks suspicious to me"
```

The first line retrieves the computation parameters of the property `D_GIF` as keyword/value pairs. These are read into the array variable **`gifparams`**, and may subsequently be accessed as **`$gifparams(format)`**, **`$gifparams(height)`**, etc. The second example shows how to attach a comment to a property value.

dataset min

```
dataset min dhandle propertylist ?filterset?
d.min(property=,?filters=?)
```

Get the minimum value of one or more properties from the elements in the dataset. The property argument may be any property attached to dataset sub-elements, or minor objects thereof. If the *filterset* argument is specified, the minimum value is searched only for objects which pass the filter set.

Examples:

```
dataset min $dhandle E_WEIGHT
dataset min [list $ehandle1 $ehandle2] A_SIGMA_CHARGE carbon
```

The first example finds the smallest molecular weight in the dataset. The second example finds the smallest (most negative, or smallest positive) Gasteiger partial charge on any carbon atom in the two argument ensembles, which form a transient dataset.

dataset molfile

```
dataset molfile dhandle ?filterset?
d.molfile(?filters=?)
```

Return the handle or reference of the *molfile* object associated with the dataset as backing page file. If no such file object exists, an empty string (for **Tcl**) or None (for **Python**) is returned.

Example:

```
set fh [dataset molfile $dh]
set fh [dataset get $dh pagefile]
```

The two commands are equivalent.

dataset move

```
dataset move dhandle datasethandle|remotehandle ?position?
d.move(target=,?position=?)
```

Move, depending on the acceptance flags of the destination dataset, either the objects in the dataset or transient dataset into another local or remote dataset, or move the dataset itself. If the destination dataset handle is an empty string (or **None** for **Python**), the dataset objects are removed from the original dataset, but not moved into any other dataset. If the destination dataset accepts datasets as members, which is not the default (see the *accept* attribute in the section on **dataset set**) the dataset is directly moved as object. Otherwise, its contained objects are moved, under preservation of the object order from the source dataset, and the source dataset is emptied, but not deleted.

Optionally, a position in the new dataset for the first moved object may be specified. This parameter is either an index (beginning with 0), or *end*, which is the default. If the contents of a dataset are spliced into another at a specific position, objects after the first element of the source dataset follow as a block.

Another special position value is *random* or *rnd*. This value moves to the dataset, or dataset contents, to a random position in the target dataset. Use of this mode with remote datasets is currently not supported.

In case of a transient command dataset the original dataset memberships of the dataset objects are *not* restored when the command completes.

The return value of the command is the dataset of the ensemble prior to the move operation. It is either a dataset handle/reference, or an empty string (**Tcl**) or **None** (**Python**) if it was not member of a dataset.

A dataset cannot be moved into itself.

Examples:

```
dataset move $dhandle $dhandle2 0
dataset move $dhandle {}
dataset move [ens list] [dataset create]
```

The first line moves all objects in the source dataset into the first (and following) positions in the destination dataset. The second example removes all elements from the dataset. This is often useful in order to avoid dataset member destruction with the **dataset delete** command. The final example shows how to move a set of ensembles (here: all ensembles currently defined in the application) into a newly created dataset via an intermediate, transient dataset.

```
dataset move $dhandle vioxx@server55:10001
```

This command moves all objects in the first dataset to the remote dataset on host *server55*, which listens on port 10001 and requires the pass phrase *vioxx* for access.

dataset mutex

```
dataset mutex dhandle mode
d.mutex(mode)
```

Manipulate the object mutex.

During the execution of a script command, the mutex of the major object(s) associated with the command are automatically locked and unlocked, so that the operation of the command is thread-safe. This applies to toolkit builds that support multi-threading, either by allowing multiple parallel script interpreters in separate threads or by supporting helper threads for the acceleration of command execution or background information processing.

Going beyond this automatic per-statement protection, this command locks major objects for a period of time that exceeds a single command. A lock on the object can only be released from the same interpreter thread that set the lock. Any other threaded interpreters, or auxiliary threads, block until a mutex release command has been executed when accessing a locked command object. This command supports the following modes:

- *lock*
  Increase the recursive mutex lock count on the object. The command returns the current lock count after the command, excluding the transient single-command lock.
- *reset*
  Release all persistent locks on the object, if they exist.
- *test*
  Return the current persistent lock count on the object. This excludes the transient per-command lock.
- *unlock*
  Decrease the recursive lock count on the object. The command returns the current lock count after the command, excluding the transient single-command lock. Unlocking an object which has not been persistently locked results in an error.

There is no *trylock* command variant because the command already needs to be able to acquire a transient object mutex lock for its execution.

The command returns the current lock count.

dataset need

```
dataset need dhandle propertylist ?mode? ?parameterdict?
d.need(property=,?mode=?,?parameters=?)
```

Standard command for the computation of property data, without immediate retrieval of results. In the common case of threaded computation, this starts a compute thread whose results or error status can be collected later. This command is explained in more detail in the section about retrieving property data.

If the dataset is not transient, the return value is the original dataset handle or reference.

Example:

```
dataset need $dhandle D_GIF recalc
```

dataset networks

```
dataset networks dhandle ?filterset? ?filtermode? ?recursive?
d.networks(?filters=?,?mode=?,?recursive=?)
```

Return a list of the handles or references of all the networks in the dataset. Other objects (ensembles, reactions, datasets, tables) are ignored. The object list may optionally be filtered by the filter list, and the result further modified by a standard filter mode argument.

If the *recursive* flag is set, and the dataset contains other datasets as objects, networks in these nested datasets are also listed.

Example:

```
set n [dataset networks $dhandle {} count]
```

dataset new

```
dataset new dhandle propertylist ?filterset? ?parameterdict?
d.new(property=,?filters=?,?parameters=?)
Dataset.New(items,property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **`dataset get`** command. The difference between **`dataset get`** and **`dataset new`** is that the latter forces the re-computation of the property data, regardless whether it is present and valid, or not.

The **Python** class method is a one-shot command. The transient dataset created from the initialization items is automatically deleted when the command finishes.

dataset nget

```
dataset nget dhandle propertylist ?filterset? ?parameterdict?
d.nget(property=,?filters=?,?parameters=?)
Dataset.Nget(items,property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading object data and attributes. It is explained in more detail in the section about retrieving property data.

For examples, see the **`dataset get`** command. The difference between **`dataset get`** and **`dataset nget`** is that the latter always returns numeric data, even if symbolic names for the values are available.

The **Python** class method is a one-shot command. The transient dataset created from the initialization items is automatically deleted when the command finishes.

dataset nnew

```
dataset nnew dhandle propertylist ?filterset? ?parameterdict?
d.nnew(property=,?filters=?,?parameters=?)
Dataset.Nnew(items,property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading object data and attributes. It is explained in more detail in the section about retrieving property data.

For examples, see the **`dataset get`** command. The difference between **`dataset get`** and **`dataset nnew`** is that the latter always returns numeric data, even if symbolic names for the values are available, and that property data re-computation is enforced.

The **Python** class method is a one-shot command. The transient dataset created from the initialization items is automatically deleted when the command finishes.

dataset nitrostyle

```
dataset nitrostyle dhandle style
d.nitrostyle(style=)
```

Change the internal encoding of nitro groups and similar functional groups in the ensembles and reactions in the dataset. Possible values for the style parameter are:

- *asis* No change
- *ionic* Change to encoding to a positive charge on the center atom, and a negative on one of the oxygens
- *xionic* As above, but also change the encoding of azides, etc.
- *neutral* Change the encoding to the neutral form with extended valence. *pentavalent* is an alias.
- *xneutral* As above, but also change the encoding of azides, etc.

The command returns the dataset handle or reference.

dataset objects

```
dataset objects dhandle ?pattern?
d.objects(?pattern=?)
```

This is a non-standard cross-referencing command. The result is a list of all the objects in the dataset, where each result list element is a list or tuple consisting of the object type (*ens*, *reaction, table, network, dataset*), and the object handle or reference. Optionally, the dataset objects may be filtered by the pattern argument which applies to the object handle.

Example:

```
dataset objects $dhandle ens*
```

is roughly equivalent to

```
dataset ens $dhandle
```

except that the latter only lists the ensemble handles, not pairs of object class name and handle.

dataset pack

```
dataset pack dhandle ?maxsize? ?requestprops? ?suppressedprops? ?compressionlib?
d.pack(?maxsize=?,?requestprops=?,?suppressedprops=?,?compressionlib=?)
```

Pack the dataset and all objects it contains into a base-64 encoded, compressed string as a serialized object. The string does not contain any non-printing characters, quotation marks or other problematic characters and is thus well suited for storage in database tables and similar applications. These packed strings are portable and platform-independent.

The maximum size of the object string (default -1, meaning unlimited) can be configured by the optional *maxsize* parameter. The size is specified in bytes. If the pack string would be longer than the maximum size, an error results.

The two optional parameter lists allow to request a specific property set to be part of the package, even if it normally would not be included, and to explicitly omit properties from the dump. No property computation is performed, and suppressed properties are not purged from the source ensemble.

The default compression library is *zlib*. Other useful variants include *lzo* and *gzip* (and there are other internal types), but these may not be available on all builds due to license issues, and you need to specify the compression library when a dataset is unpacked. It is generally recommended to stay with *zlib*.

The return value of this command is the packed string.

In **PYTHON**, datasets support the standard *pickle*/*unpickle* protocol.

Example:

```
dataset pack $dhandle
```

dataset pop

```
dataset pop dhandle|remotehandle ?position? ?timeout?
d.pop(?position=?,?timeout=?)
Dataset.Pop(dref/remotehandle,?position=?,?timeout=?)
```

Remove an object from a dataset. The handle or reference of the selected object is returned, and the object is no longer a member of the dataset when the command completes. If a timeout is specified, it is transferred to the dataset attribute of the same name before the command is executed, as with a **`dataset set`** command.

By default the first object in the dataset, at index zero, is returned. A different object can be selected by means of the optional position argument. It can be a numerical index, *end* for the last object, *rnd*/*random* for a random selection. If the object index if larger than the maximum index of any object, it is silently rewritten to *end*. Random pops are not supported on remote datasets.

This command works with remote datasets. In that case, the object is transferred via an intermediate serialized object representation over the network. It is unpacked on the local interpreter, and deleted on the remote interpreter.

If the desired dataset object cannot be found, and a timeout is set, including a negative value for an unlimited wait time, the command suspends execution until the object appears in the dataset, for example from a different script thread or as result of a remote object insertion. If a wait would be executed, but the *eod/targeteod* parameter pair of the dataset indicate that no further data can be expected, the command returns an empty string (for **TCL**) or **`None`** (for **PYTHON**) instead of the object handle or reference, but does not trigger an error. Otherwise, if the object cannot be delivered immediately or after the timeout, an error results.

Example:

```
set eh [dataset pop $eh end]
```

dataset properties

```
dataset properties dhandle ?pattern? ?intersectionmode?
d.properties(?pattern=?,?intersectionmode=?)
```

Get a list of valid properties of the dataset proper and the dataset objects. By default, both dataset properties (prefix `D_`) as well as the properties of the objects in the dataset (prefix `E_` for ensembles, `X_` for reactions, `T_` for tables, `N_` for networks, `D_` for datasets as members) and the properties of their minor objects (atoms, bonds, etc.) are listed. Property subsets may be selected by specifying a string filter pattern. In case of dataset element properties which are not present in all dataset members, the default intersect mode is *union*, meaning that all properties are reported for which at least a single instance in any member exists. The alternative mode *intersect* only lists those dataset member properties which are present at all dataset members.

This command may also be invoked as **`dataset props`** or **`d.props()`**.

Example:

```
dataset properties $dhandle D_*
dataset props $dhandle E_* intersect
```

The first example returns a list of the currently valid dataset-level properties. The second example lists ensemble properties which are present in all dataset objects.

dataset purge

```
dataset purge dhandle propertylist ?emptyonly?
d.purge(?properties=?,?emptyonly=?)
```

Delete property data from the dataset. The properties may be both dataset properties (prefix `D_`) or properties of the dataset members, such as ensemble or atom properties. If a property marked for deletion is not present on an object, it is silently ignored.

If an object class name, such as *ens* or *atom*, is used instead of a property name, all properties of that class set on the objects in the dataset are deleted, if they are not locked, or filtered out by the optional empty-only flag.

Besides normal property names, a few convenient alias names for common property deletion tasks of ensembles in a dataset, or the reaction ensembles of reactions in the dataset, are defined and can be used as a replacement for the property list. These include:

- *atomquery*
  Delete atom query information.
- *atomstereochemistry*
  Delete all atomic atom stereo descriptors, but keep those for bonds.
- *bondquery*
  Delete bond query information.
- *bondstereochemistry*
  Delete all bond stereo descriptors, but keep those for atoms.
- *dataset*
  Delete all dataset properties (prefix D_).

- *ens*
  Delete all ensemble properties on the dataset objects (prefix E_).
- *isotopes*
  Delete isotope information in `A_ISOTOPE` and other isotope properties which may be defined in future software versions.
- *query*
  Delete atom and bond query information.
- *radicals*
  Delete atomic radical information in `A_RADICAL` and other radical-related properties which may be defined in future software versions.
- *reaction*
  Delete all reaction properties on the dataset objects (prefix X_).
- *stereochemistry*
  Delete all stereochemistry descriptors, including 2D wedges, but not 3D coordinates. The implicit property list includes `A_LABEL _STEREO, B_LABEL_STEREO, A_CIP_STEREO, B_CIP_STEREO, A_DL_STEREO, B_CISTRANS_STEREO, A_HASH_STEREO, B_HASH_STEREO, A_MAP_STEREO`, `B_MAP_STEREO`, `A_STEREOINFO`, `B_STEREOINFO`, `A_STEREO_GROUP`, `M_STEREO_COUNT`, `E_STEREO_COUNT` and `B_FLAGS` (only wedge bits, the property remains valid if present).
- *wedges*
  Delete wedge bond flags in property `B_FLAGS`. If `B_FLAGS` is not present, the command is ignored and no computation attempt is made.

The optional boolean flag *emptyonly* restricts the deletion to those properties where all the values for a property associated with a major object (such as on all atoms in an ensemble for atom properties, or just the single ensemble property value for ensemble properties) are set to the default property value.

The return value is the original dataset handle or reference.

Examples:

```
dataset purge $dhandle D_GIF
dataset purge [ens list] E_IDENT 1
dataset purge $dhandle stereochemistry
```

The first example deletes the property data `D_GIF` for the selected dataset if it is present. The second example deletes property `E_IDENT` from all ensembles in the current application if their property value is equal to the default value of `E_IDENT`. The third examples removes stereochemistry from all dataset ensembles.

dataset reactions

```
dataset reactions dhandle ?filterset? ?filtermode? ?recursive?
d.reactions(?filters=?,?mode=?,?recursive=?)
```

Return a list of all the handles or references of the reactions in the dataset. Other objects (ensembles, tables. datasets, networks) are ignored. The object list may optionally be filtered by the filter list, and the output further modified by a standard filter mode.

If the optional boolean *recursive* argument is set, reactions of which ensembles in the dataset are a component are also listed. Furthermore, if the dataset contains datasets as elements, these are recursively traversed, and reactions in these, as well as reactions as components of ensembles in these datasets, are listed. If the output mode of the command is a handle list, items found by recursion are appended in a straight fashion, without the creation of nested lists. By default the recursion flag is off. Regardless of the flag value, reactions which are associated with rows of a table in the dataset, but are not themselves dataset members, are not output.

Example:

```
set xlist [dataset reactions $dhandle]
```

Return a list of the handles of the reactions in the dataset.

```
set cnt [dataset reactions $dhandle {} count 1]
```

returns a count of all reactions which are either directly members of the dataset, or indirectly because ensembles in the dataset are part of a reaction, or which are contained in datasets which are a themselves a member of the primary dataset.

dataset read

```
dataset read dhandle ?datasethandle/enshandle? ?#recs|batch|all?
d.read(?target=?,?limit=?)
```

This command returns handles or references of *duplicates* of one or more objects from the current dataset iterator position (*record* attribute). Its arguments mimic those of the **`molfile read`** command. The iterator record attribute is automatically incremented. When the end of the dataset is reached, an empty result is returned, but no error is raised.

The return value is usually the handle or reference of the object duplicated from the dataset member at the current read position. If an optional target dataset has been specified. the object is appended to that dataset, and the return value is the target dataset handle. It is also possible to use the magic dataset handles *new* or *#auto*, which create a new receptor dataset.

If instead of a target dataset an existing target ensemble is specified, the recipient ensemble is cleared, and the read dataset object placed into its hull without changing its handle. This requires that the read object is an ensemble, and not a reaction, table, dataset or network, and that only a single item is read. It is also possible to use an empty argument to skip these options.

By default, a single object is duplicated and the iterator record attribute of the dataset incremented by one. With the optional third argument, a different number of objects can be selected for reading as a block. The special value *all* reads all remaining objects, and *batch* copies a number of objects corresponding to the *batchsize* dataset attribute. If there are insufficient objects in the dataset to read all requested records, only the available set is returned, and no error results.

The dataset contents are not changed by this command. All extracted items are object duplicates. In order to fetch original objects from the dataset, use the **`dataset pop`** command, or the various object **`move`** commands.

The command variant **`dataset hread`** provides the same functionality as this command, but additionally adds a standard set of hydrogen atoms to the duplicates.

dataset ref

```
Dataset.Ref(identifier)
```

Python only method to get a dataset reference from a handle or another identifier. For datasets, other recognized identifiers are dataset references, integers encoding the numeric part of the handle string, the dataset `UUID` or name, or a table handle (which returns the dataset embedded in the table).

dataset remove

```
dataset remove dhandle ?handle?...
d.remove(?handle/ref?,...)
```

Remove objects from a dataset. The removal objects must be in the dataset.

If the dataset is not virtual, the command returns the dataset handle or reference.

dataset rename

```
dataset rename dhandle srcproperty dstproperty
d.rename(srcproperty=,dstproperty=)
```

This is a variant of the **`dataset assign`** command. Please refer the command description in that paragraph.

dataset request

```
dataset request dhandle propertylist ?reload? ?modelist?
```

Request property data for a dataset when the dataset is not maintained locally, but a partial shadow copy of a remotely managed dataset. It is assumed to have been only partially transferred via **RPC** to a slave from a master controller application, for example for display purposes, but without the full data content, which resides on the master.

If the requested property data is already present on the slave, and the *reload* flag is not set, this command is equivalent to a **`dataset need`** command and does not invoke communication with the master. Otherwise, the master is asked to provide the information, which may be calculated on the master only after receiving the request, or even delegated by the master to another remote server for computation.

Once the requested data has been received by the slave, it is added to the property data set of the local dataset copy. The optional *modelist* parameter is the same as in the **`dataset need`** command. This command is used to guarantee that critical or non-computable property data is obtained from the master. Local, unsynchronized data may still be computed by the slave using standard property data access commands. It is currently not possible to send data back to the master.

This command is only available on toolkit versions which have been compiled with RPC support.

In the absence of errors, the command returns a boolean status code. If it is zero, the request failed in a non-critical way. This for example happens in case the dataset is not under control of a remote application.

Example:

```
if {![dataset request $dhandle A_XY]} {
   dataset need $dhandle A_XY
}
```

is a bullet proof method of guaranteeing that correct atomic 2D display coordinates are present for the dataset structures even if the script is run in a master/slave context.

This command is not supported in the **Python** interface.

dataset rewind

```
dataset rewind dhandle
d.rewind()
```

Reset the dataset iterator record. This is equivalent to setting the *record* attribute to one.

dataset scan

```
dataset scan dhandle expression/queryhandle ?mode? ?parameterdict?
d.scan(query=,?resultmode=?,?parameters=?)
Dataset.Scan(items,query=,?resultmode=?,?parameters=?)
```

Perform a query on the dataset or transient dataset. The syntax of the query expression is the same as that of the **`molfile scan`** command and explained in more detail in its section on query expressions. Essentially, this command behaves like an in-memory data file version of the **`molfile scan`** command. However, currently queries work on ensembles and reactions as dataset members only. Any table, network or other object which is a member of a scanned dataset is skipped. Skipped items still count as records for positioning and query result output. In the absence of a specified scan record list (*order* parameter), dataset scans begin at the current position of the iterator *record* attribute that is shared with the **`dataset read/hread`** commands.

The optional parameter dictionary is the same as for **`molfile scan`**, but not all parameters are actually used. At this time, only the *matchcallback, maxhits, maxscan, order, progresscallback, progresscallbackfrequency, sscheckcallback, startposition* and *target* parameters have an effect. If result ensembles or reactions are transferred to a remote dataset via the *target* parameter, they are not deleted from the local dataset but duplicates are created instead. This is because the original objects are members of the dataset which, just like a structure file would, should remain unchanged as result of a scan. In contrast, in file scans, the transferred ensembles and reactions were read from file and created as new objects during the scan, and sending these does not change the underlying file. In case a progress callback function is used, the dataset handle is passed as argument in place of the *molfile* handle in **`molfile scan`**.

The return value depends on the mode. The default mode is *enslist*. The following modes are supported for dataset queries:

- *array* (or alias *tclarray, dict, pythondict*)
  The mode parameter is a list consisting of the mode selector *array* and a nested list of properties and pseudo-properties. Each property item can be a list of one to three elements. The first element is a property or pseudo-property, the second element a name, and the third element again a property or pseudo property. The the second property item list element is omitted, the name is the same as the first element. If the third element is missing, it is assumed to be the pseudo-property *record*.

  In this mode, the command returns a list of the names of the created arrays. For each name, a global **Tcl** array variable or **Python** dictionary is created, and for each match, a **Tcl** array element with an element name equal to the value of the first item specification index and an element value equal to the value of the third item specification is created (or a dictionary entry with key and value for **Python**). For example, the scan mode specification

  ```
  {array {E_NAME name2rec} {record rec2name E_NAME}}
  ```

results in the creation of two global **Tcl** arrays or **Python** dictionaries in the current interpreter, called *name2rec* and *rec2name*. The first has array elements (for **Python**, dictionary keys) where the element name is the name of the matching structure (property `E_NAME`), and the value the pseudo-record number (because it is the default). The second array has elements where the record number is the array element name, and the corresponding value the structure name. The return value of the scan statement is the list (tuple for **Python**) *"name2rec rec2name"*, containing the names of the two variables created.

If array or dictionary elements for a specific key already exist, the new value is appended as a list or tuple object. The result registration procedure does not overwrite the existing content. So, for example in above case, if there are multiple records with the same structure name, the array element indexed by name would contain a list or records, not just a single record. Since the global arrays or dictionaries are persistent, data is also appended over multiple scan statements. If this is not desired, a statement like **`unset -nocomplain $arrayname`** should be executed before the scan is started. It is legal to use the same array or dictionary name for the registration of multiple properties. In this case, each match appends a new list element for every reported property, though these lists will not be nested.

- *bitvector*
  Return a string-encoded bit vector (series of 0s and 1s) indicating the match status for every visited record.
- *boolean*
  Return a boolean value indicating whether the next record matches or not.
- *booleanvector*
  Return a boolean vector (series of 0s and 1s as vector elements) indicating the match status of every visited record. The difference to the *bitvector* mode is that in the scripting interface the vector elements are already isolated elements, for example they appear space-separated in the string form.
- *count*
  Count the number of hits. The result value is an integer.
- *delete*
  Delete hits from the dataset. This is the only scan command which actually changes the dataset.
- *ens*
  Return the handle of the first matching ensemble. The query is stopped at that point. If no hits are found, an empty string is returned. If a local target dataset is specified, a found ensemble is removed from the scanned dataset.
- *enslist*
  Return the handles of all matching ensembles. If no hits are found, an empty list is the result. If a local target dataset is specified, the found ensembles are removed from the scanned dataset.
- *exists*
  A boolean check for the existence of a match. The same as *count*, except that the scan stops after the first match.

- *index*
  Return the positional index of the first matching dataset object. This is the same as the *record* mode value minus one.
- *indexlist*
  Return the positional indices of the matching dataset objects. This is the same as the *recordlist* mode values minus one.
- *molfile*
  The mode parameter is a list consisting of the mode selector *molfile* and a structure file handle, which must have been opened for writing, appending, or updating. The first matching structure is written to the file, and the command stops at that point.The output file handle attributes determine format, selection of data written, structure encoding conventions such as hydrogen status, etc. If no matching structure is found, nothing is written. In this mode, the return value of the command is the matching record number of the input file, just as in the *record* mode.
- *molfilelist*
  The mode parameter is a list consisting of the mode selector *molfilelist* and a structure file handle, which must have been opened for writing, appending, or updating. Matching structures are written to that file. The output file handle attributes determine format, selection of data written, structure encoding conventions such as hydrogen status, etc. If no matching structures are found, nothing is written. This mode is also implicitly selected if a structure file handle is directly provided as mode argument. In this mode, the return value of the command is a list of the matching record numbers of the input file, just as in the *recordlist* mode
- *property*
  The mode parameter is a list consisting of the mode selector *property* and a sequence of properties and pseudo-properties. The selected properties for the first match are returned as a list, and the command stops at that point. If there are no hits, an empty string is returned.
- *propertylist*
  The mode parameter is a list consisting of the mode selector *propertylist* and a sequence of properties and pseudo-properties. The selected properties for all matches are returned as a nested list. If there are no hits, an empty string is returned. This mode is also selected if the mode argument is simply a list of property and pseudo property names without an identifiable mode keyword as first list element.
- *reaction*
  Return the handle of the first matching reaction. The query is stopped at that point. If no hits are found, an empty string is returned. If a local target dataset is specified, a found reaction is removed from the scanned dataset.
- *reactionlist*
  Return the handles of all matching reactions. If no hits are found, an empty list is the result. If a local target dataset is specified, the found reactions are removed from the scanned dataset.
- *record*
  Return the object sequence number of the first hit. Sequence numbers begin, for the sake of comparability with structure file scan record numbers, with one.

- *recordlist*
  Return object sequence numbers of all hits, or an empty list. Sequence numbers begin, for the sake of comparability with structure file scan record numbers, with one.
- *table*
  The mode parameter is a list consisting of the mode selector *table* and a sequence of properties and pseudo-properties. This scan mode returns a table handle. The table is automatically configured with properly typed columns corresponding to the requested properties. For each hit, a row is added. If there are no hits, a table handle is still returned, but the table does not have any rows. This retrieval mode is only available if the toolkit has been compiled with table support. The individual properties may also be specified each as a list consisting of the property name, and an arbitrary string. In that case, the string is used as the column name. By default, the column names are the same as the name of the property they store. Example:

  ```
  {table {E_NAME name} {E_CAS casno} record}
  ```

  sets up a table with three columns called *name*, *casno* and *record*. The first two columns contain property data from the matching file records, the last one the record in the file which matched.

  Instead of the keyword *table*, an existing table handle may also be used. In that case, any existing matching table columns are automatically re-used to store result data. Additionally specified properties are added as new columns to the right of the previously existing columns. New table rows generated by matches are appended to the bottom of the table.
- *tablecollection*
  Since all objects are already in memory, this mode is identical to the *table* scan mode for dataset scans. No table reference object duplicates are created. The result table always refers the dataset objects directly.
- *vrecord*
  For dataset scans, this is the same as *record*.
- *vrecordlist*
  For dataset scans, this is the same as *recordlist*.

If requested property data is not present on the matched dataset objects, an attempt is made to compute it. If this fails, the table object in retrieval mode *table* contains **`NULL`** cells, and property retrieval as list data produces empty list elements, but no errors. For minor object properties, the property list retrieval modes produce lists of all object property values instead of a single value. In *table* mode, only the data for the first object is retrieved, which makes this mode less suitable for direct minor object property retrieval.

The following pseudo properties can be retrieved in addition to normal properties:

- *avgscore*
  The average value of all computed scores, such as Tanimoto, Cosine or Tversky similarity scores, in the matching query for this result.
- *conformerindex*
  The index of the matching conformer in case of 3D queries with multiple conformations, -1 if no matching conformer index was determined.

- *conformer*
  A list of the atomic coordinates of the matching conformer, if a 3D query was performed. If this is not the case, an empty vector is the result. The data type of this vector is *coorvec* (x,y,z-triples as vector elements).
- *filename*
  This property is only provided for compatibility with **`molfile scan`**. It is always an empty string in this command.
- *index*
  The object sequence index of the matching object. For datasets, this is the same as the *record* value minus one.
- *image*
  A structure **GIF** image (property `E_GIF`) with highlighted matching substructure atoms and bonds. A normal `E_GIF` retrieval property would just show the structure, but without highlighting. The data type of this property is the same as that of `E_GIF` (depending on the configuration, a *diskfile* reference or an in-memory *blob*).
- *matchatoms*
  An integer vector holding the labels of all atoms matching the substructures used in evaluating the query expression. If no substructure was used for the match, this vector is empty. *highlighatoms* is an alias for this pseudo property.
- *matchbondatoms*
  The same as *matchbonds*, except that each element is a pair of the labels of the matching atoms in the bonds, not the bond label as a single number.
- *matchbonds*
  An integer vector holding the labels of all bonds matching the substructures used in evaluating the query expression. If no substructure was used for the match, this vector is empty. *highlightbonds* is an alias for this pseudo property.
- *matchcount*
  The first element of the *matchcounts* array, as described below. If the query does not contain any substructure match nodes, the result is empty.
- *matchcounts*
  An integer vector holding the number of distinct substructure matches for substructure query nodes in the query tree. For normal substructure expressions, this value can only be zero or one because the standard substructure match mode only checks for the presence of any match (match mode *first*). Additionally, this value can be minus one if the node was never evaluated, for example because it is part of an *or* expression. Only if the *count* modifier is used together with the substructure query operator, or the substructure operator is the range operator, the possibility of multiple matches is evaluated and larger values can be obtained. For these operations the default match mode is *distinctinneratoms* (see **`match ss`** command).
- *matchmask*
  A bitvector indicating which children of the root query node have matched. The length of the bitvector is the same as the number of children of the root node. A typical application of this retrieval item is in combination with a *range* node as root node.

- *maxscore*
  The maximum value of all computed scores, such as Cosine, Tanimoto or Tversky similarity scores, in the matching query for this result.
- *merit*
  For queries which use a merit/demerit rating scheme (for example, Bruns/Watson queries) this retrieves the accumulated merit/demerit sum of the top-level query node. The query needs to match for this retrieval to work, so in case none of the demerit rules match, you get an empty result, not a default zero merit/demerit value. Internally, there is no distinction between merit and demerit scores. The keyword *demerit* is an alias for this pseudo-property.
- *minscore*
  The minimum value of all computed scores, such as Cosine, Tanimoto or Tversky similarity scores, in the matching query for this result.
- *parent*
  The parent structure of the matching structure as a packed, base64-encoded serialized object string. If the dataset ensemble does not already contain it, it is computed from the structure as property `E_PARENT_STRUCTURE`.
- *productmatchatoms*
  The same as the *matchatoms* pseudo property, but for the ensemble on the right side of a matching reaction, not a simple structure. If no reaction was matched, this is an empty list.
- *productmatchbondatoms*
  The same as the *matchbondatoms* pseudo property, but for the ensemble on the right side of a matching reaction, not a simple structure. If no reaction was matched, this is an empty list.
- *productmatchbonds*
  The same as the *matchbonds* pseudo property, but for the ensemble on the right side of a matching reaction, not a simple structure. If no reaction was matched, this is an empty list.
- *queryid*
  The ID of the search tree query item which was responsible for the principal match. Every tree element of a query expression possesses an ID, starting with 1, and then assigned in incremental sequence from left to right in depth-first manner. For simple property or structure match expressions, the query ID is the ID of the matching branch, i.e. one for single-node expressions. For logical expressions with an *or*, *orcontinue* or *not* node, the overall reported query ID is that of the first matching leaf node. For expressions, where all leaves need to be checked, the query ID is the ID of the *and* or *eor* node where all leaves matched, not the ID of any individual leaf node.
- *reagentmatchatoms*
  The same as the *matchatoms* pseudo property, but for the ensemble on the left side of a matching reaction, not a simple structure. If no reaction was matched, this is an empty list.
- *reagentmatchbondatoms*
  The same as the *matchbondatoms* pseudo property, but for the ensemble on the left side of a matching reaction, not a simple structure. If no reaction was matched, this is an empty list.

- *reagentmatchbonds*
  The same as the *matchbonds* pseudo property, but for the ensemble on the left side of a matching reaction, not a simple structure. If no reaction was matched, this is an empty list.
- *record*
  The record number. In the context of in-memory datasets, this is the dataset object list index of the matching object plus one. *rc* is an alias for this pseudo property. Use the *index* attribute to directly obtain the dataset index.
- *rgatoms(rg)*
  A list of the atom labels in a matching structure which were mapped to an expanded R-group atom in the query. The property index is the name of the R-group of interest defined in the substructure, usually something like *R1*. If there was no expanded R-group of that name, the result list is empty.
- *rgattachments(rg)*
  A nested list of the atom label pairs of the bonds in a matching structure which connect between the structure framework and the atoms expanded as the named R-group *rg*. If there was no expanded R-group of that name, the result list is empty.
- *score*
  The first element of the *scores* array, as described below. If the query does not contain any scoring expressions, the result is empty.
- *scores*
  An integer vector of the results of all query expression branches, in depth-first left-to-right order, which computed a score, such as structure similarity queries with Cosine, Tanimoto or Tversky bitvector comparisons. In case a branch was not evaluated when the match was determined, zero is returned.
- *structure*
  The dataset structure as a packed, base64-encoded serialized object string.
- *vrecord*
  For dataset scans, this is always the same as *record*.

These pseudo properties are identical to those available for structure file queries. However, structure file queries support a couple of additional pseudo properties which are not available for dataset queries.

The **Python** class method is a one-shot command. The transient dataset created from the initialization items is automatically deleted when the command finishes.

Examples:

```
dataset scan $dhandle {E_WEIGHT < 200} recordlist
dataset scan $dhandle "structure >= c1ccccc1" {table E_NAME E_LOPG record}
dataset scan $dhandle "structure >~ $sshnd 90" {cmpvalue E_REACTION_ROLE X_IDENT}
```

The first example returns the record numbers (dataset member indices plus one) of all structures in the dataset which have a molecular weight of less than 200.

The seconds example generates a table with columns for name, logP and record number. The table is filled with data from all structures which contain a phenyl ring as substructure.

The final example returns a nested list of the properties of all dataset structures which have a Tanimoto similarity of 90% or more to the structure which is represented by its handle stored in the variable **`$sshnd`**. In this example, the ensembles are expected to be also part of a reaction, which is possible since reaction and dataset membership are completely unrelated. Each result list element contains the actual similarity value (which is the only comparison result value with a threshold evaluated in the query, so there is no ambiguity which comparison result *cmpvalue* refers to), the role of the ensemble in the reaction (*reagent*, *product*, *catalyst*, etc.) from property `E_REACTION_ROLE`, and the reaction ID in `X_IDENT`. The scan mode is here automatically set to *propertylist*, because the mode list consists exclusively of names of properties and pseudo properties.

Another example:

```
set is_chno [dataset scan $ehandle {formula = C0-H0-N0-O0-} count]
```

This command checks whether the ensemble (which is, for the duration of the command, embedded into a transient dataset) contains only elements C, H, N and O.

dataset set

```
dataset set dhandle ?property value?...
d.set(?property,value?,...)
d.set({property:value...})
d.property = value
d[property] = value
```

Standard data manipulation command. It is explained in more detail in the section about setting property data.

In addition to property data, the dataset object possesses a few attributes, which can be retrieved with the **`get`** command (but not by its related sister subcommands like **`dget`**, **`sqlget`**, etc.). Many of them are also modifiable via **`dataset set`**. These attributes are:

- *accept*
  A bit set indicating the object classes the dataset accepts as members. Currently, this can be any combination of *ens*, *reaction*, *table*, *network* and *dataset*. The default acceptance mask is the union of all *ens*, *reaction* and *table*, excluding datasets and networks as allowed dataset objects. If an attempt is made to add an unacceptable object to a dataset, the command (such as **`ens move`**, **`dataset add`**, etc.) throws an error. If the object added to a dataset is a dataset, but the dataset does not accept datasets as members, the objects contained in the source dataset are added instead.
- *address_city*
  The city part of the author contact address.
- *address_country*
  The country part of the author contact address, following the ISO3166 standard.
- *address_state*
  The state part of the author contact address. Empty if not applicable.
- *address_street*
  The street address part of the author contact address. Includes floor, house number, etc.
- *address_zip*
  The **ZIP** code or other applicable postal code of the author contact address.

- *affiliation*
  The institution the author works for.
- *affiliationduns*
  The **DUNS** registration ID of the affiliated institution. This is primarily useful for US government projects.
- *affiliationurl*
  The **URL** of the affiliated institution.
- *author*
  The author of the dataset, as free-form string data.
- *authorurl*
  A **URL** with information on the author of the dataset, or an empty string if unset.
- *batchsize*
  The number of objects in the dataset which form a batch. This can for example be used in the **`dataset read`** command. The default batch size is 10.
- *category*
  A category string to be used if the dataset is stored in a repository.
- *classuuid*
  The base class **UUID** of this dataset object, as associated with its authorship attributes.
- *coords*
  f the toolkit was compiled with factory support, these are the coordinates of the object icon on its workbench, encoded as integer pair. This attribute can be changed.
- *counter*
  An integer counter which is automatically incremented every time an object is moved into the dataset, but not when the object only changes its position within the dataset. It can also be reset to an arbitrary value, and later dataset additions increment the counter from that user-specified value. It is not decremented when objects leave the dataset, so this attribute is not necessarily the same as the dataset size. The initial counter value at dataset object creation time is zero. Depending on its mode, this attribute may interact with the *insertcontrol* attribute.
- *datasetcount*
  A read-only attribute reporting the number of dataset objects currently contained in the dataset.
- *date*
  The date the dataset was defined.
- *deletable*
  A boolean flag indicating whether the dataset can be deleted at this time or not. This is a read-only attribute. Under certain circumstances, such as a pending **`dataset wait`** command, or the use of the dataset object as argument to a scripted computation function expecting to be able to set function result data as property values, the dataset is marked as undeletable and any destruction command will silently fail.
- *deselection*
  The inverse of the *selection* attribute, i.e. get all unselected object indices, or set the selection by providing a list of object indices which are not selected.

- *doi*
  A digital object identifier for the dataset object content, if defined.
- *email*
  A contact email of the author of the dataset.
- *enscount*
  A read-only attribute reporting the number of ensemble objects currently contained in the dataset.
- *eod*
  The value of the end-of-data marker. This attribute is typically used in multi-threaded applications to indicate that feeder threads have exhausted their data supplies and that no further dataset objects are expected to arrive in the dataset. This attribute is internally used by the **`dataset pop`** and **`dataset wait`** commands to determine whether they should continue to wait or exit with an empty result. The initial value of this attribute is zero.
- *eodcheck*
  Perform a check whether at least one object is in the dataset, or is expected to arrive later. If objects are currently in the dataset, or the eod attribute value is less than the *targeteod* attribute value, the command returns zero, otherwise one. This check is not reliable for remote datasets.
- *failures*
  A list of properties for which computation failed on this object. This is a read-only attribute. Depending on configuration settings, this information may be used to block pointless attempts at re-computation of incomputable data.
- *footer*
  If the toolkit was compiled with factory support, this is the footer of the object icon on a workbench. This attribute can be changed.
- *gflags*
  If the toolkit was compiled with factory support, this is the currently set object icon rendering flag collection.
- *header*
  If the toolkit was compiled with factory support, this is the header of the object icon on a workbench. This attribute can be changed.
- *hidden*
  Flag indicating whether the object is hidden. This is not the same as the invisible state. This attribute is intended to be used for rendering selections. This attribute can be changed.
- *highwatermark*
  An integer specifying a high water mark object on the dataset. Some commands use this attribute for automatic start or cancellation of operations until the object count has decreased to the low water mark, or for automatic start of processing services until the low watermark has been reached again. The default high watermark value is one. The **`dataset wait`** command uses this threshold as default command parameter.

- *invisible*
  Flag indicating whether the object is invisible. This is not the same as the *hidden* state. An invisible object is no longer accessible via its handle. This is usually the case for objects which are scheduled for deletion, but still have lingering pointer references. This attribute is read-only.

- *insertcontrol*
  This parameter controls what happens when an attempt is made to add another object into a dataset. The default mode is *add*, which means that the object is inserted in the database if there is no size control active, or room can be made by waiting for already inserted objects to be removed (see *sizecontrol* parameter). Otherwise, in that mode an error results.

  Additional insertion control modes are *disabled* (all insertions into the dataset are blocked), *discardfirst* (if the maximum size has been reached, delete first object in dataset to make room), *discardlast* (if the maximum size has been reached, delete last object in dataset to make room), *discardobject* (if the maximum size has been reached, delete the object to be inserted), *discardalways* (never attempt an actual insertion, always delete the insertion object), *ignore* (if insertion cannot be performed, leave the insertion object where it currently is, with preservation of current dataset membership) and *unlink* (silently remove the insertion object from its old dataset, if it is a member of one, but do not insert it into the target dataset if that would exceed its maximum size).

  If the object cannot be inserted and is deleted (but not if it is just unlinked or ignored, and thus continuing to exist) the dataset counter is still incremented.

  The final mode is *discardrandom*. In this mode, if the maximum size of the dataset has not yet been reached, the object is simply added. Otherwise, a random number between one and the counter attribute of the dataset is computed. If the number is larger than the maximum dataset size, the object to be inserted is deleted, as in the *discardnew* mode. If the random number is between one and the dataset size, the object in the dataset at the random position is deleted. After that, the new object inserted at its designated position, which is not necessarily the position of the removed object. This mode is intended to support convenient sampling of object subsets. The random procedure yields the same mathematical results as directly picking random objects from the total object pool passing through the dataset, but may be interrupted at any time yielding a random subset of the objects processed so far.

- *instanceuuid*
  The instance **UUID** of this dataset, as associated with its authorship attributes.
- *infourl*
  A **URL** with information on the dataset object content, or an empty string if unset.
- *keywords*
  A list of keywords associated with the table object.
- *license*
  The license class associated with this dataset object. Setting the license to a standard type updates the associated **URL** with a standard location.
- *licenseurl*
  A **URL** with details about the dataset object license.
- *literature*
  A free-form literature reference for the dataset.

- *lowwatermark*
  An integer specifying a low water mark object count on the dataset. Some commands use this attribute for automatic scheduling or termination of actions. The default low watermark is zero.

- *maxsize*
  The maximum number of objects the dataset will accept. If it is set to a negative value, which is the default, the maximum number of objects is unlimited. The effects of an attempt to overload the dataset depend on the settings of the *sizecontrol* attribute of the dataset.

- *modcount*
  The content and data modification count on the object. This is a read-only attribute.

- *mutexcount*
  The number of recursive mutex locks held for this object. Only supported on Linux.

- *name*
  A free-form dataset name as string.

- *networkcount*
  A read-only attribute reporting the number of network objects currently contained in the dataset.

- *orcid*
  The **ORCID** code of the author (see www.orcid.org).

- *pagefile*
  The handle of a *molfile* object. If this is set, the current contents of the dataset are deleted, the *pageoffset* attribute set to the current input position of the file, and a number of records up to the current value of the *pagesize* attribute are read into the dataset. If this attribute is set to an empty string, the connection between the dataset and the structure file is abolished.

- *pageoffset*
  The file record offset of the first object in the dataset, if the dataset is linked to a file. If this value is changed, and a link is active, dataset objects with file records outside the *offset/pagesize* window are deleted from the end or beginning, and new objects are added from the backing file as required.

- *pagesize*
  The number of records to keep in the dataset in case it is linked to a file. If this value is changed, and a link is active, dataset members are deleted from the dataset, or added from the backing file as necessary.

- *parent*
  Get the handle of the parent object, if the dataset is an embedded object, e.g. an integral component of a table, factory or station object. If the dataset is a standalone object, an empty string is returned. The parent attribute is not the same as dataset membership (see **`dataset dataset`** command), which can be changed (see **`dataset move`** command and the *accept* dataset attribute). This attribute is read-only. An embedded dataset object cannot be dissociated from its owner.

- *passphrase*
  A string which needs to be presented by remote interpreters if they connect to the listener port of the dataset object. An empty string is equivalent to no pass phrase.
- *path*
  The repository path for displaying hierarchical repository trees. This attribute is independent of any file system paths.
- *phone*
  A contact phone number of the author.
- *port*
  An integer port number at which a listener thread waits for connections from remote interpreters for the addition or removal of objects. If this attribute is set to an empty string, an existing listener thread is terminated and remote connections are no longer accepted.
- *progress*
  A user-defined progress value intended to track the state of lengthy operations on the table. It is an integer between zero and one hundred and is initially set to zero. When the argument is set, it accepts a floating point value, but the stored value is automatically rounded to the next integer and forced into the 0..100 range.
- *pyobject*
  If the toolkit was compiled with Python support, this attribute reports the memory address of the Python wrapper class instance, if it exists. This attribute is read-only.
- *pyrefcount*
  If the toolkit was compiled with Python support, this attribute reports the reference count of the Python wrapper class instance, if it exists. This attribute is read-only.
- *reactioncount*
  A read-only attribute reporting the number of reaction objects currently contained in the dataset.
- *record*
  The current iterator record position. The first object in the dataset corresponds to record 1.
- *refcount*
  If the **TCL** interpreter is using native Cactvs objects instead of string-based major object handles and integer-based minor object labels to identify toolkit objects, this returns the number of **TCL** object references active for this ensemble. This attribute is read-only.
- *references*
  Cross references of the dataset. This is a nested list of class **UUID**s and reference type tags.
- *regid*
  For registered datasets, the registration ID. Zero if this is a private dataset.
- *room*
  A read-only integer attribute which indicates whether the dataset has room for the insertion of another object. Datasets without size control always return 1, as do datasets which still have room for more objects. Return value 0 indicates that the maximum size has been reached, and no alternative action has been configured. Other possible special return values are -1 (insertion succeeds, but delete the inserted object), -2 (insertion will silently fail, the object remains in its old dataset membership), -3 (the object will be unlinked from any

existing dataset, but silently not inserted into the new dataset) and -4 (the object will not be inserted in the target dataset, instead an application-specific alternative action will be taken). This attribute only checks the capacity of the dataset, not whether it will reject the object because it is of an unsuitable class (see *accept* attribute). In multi-threaded applications, the status value may become outdated before an insert command on the target dataset can be executed.

- *scoped*
  A boolean object visibility control flag. If set, and global control flag `::cactvs(object_scope)` is also set, the object is visible only in the TCL interpreter which set the scope flag and thus claimed it. Object list commands executed in other interpreters omit this object, and attempts to decode its handle in other interpreters will fail. The most common use of this feature is the hiding of persistent chemistry objects in scripted property computation functions.
- *selected*
  Flag indicating whether the object is selected. This attribute can be changed. This attribute works on the dataset object proper, not its content - see the *selection* attribute below.
- *selection*
  Upon retrieval, this attribute is a list of the position indices of all objects in the dataset which have the *selected* status flag. The index begins with zero, and the result is an empty list if there are no selected objects.

  On setting, `dataset set` first clears all dataset object selections. The command `dataset append` retains it. The argument is then parsed as a list of integer object indices, and the selection flag is set for all those indices where objects can be found in the dataset. Indices outside the range between zero and the dataset size minus one or duplicate index specifications are silently ignored.

  To check or set the selection status of the dataset object proper, use the *selected* attribute.
- *size*
  Get the number of objects in the dataset. This is a read-only attribute. It is equivalent to the `dataset count` command without any filters.
- *sizecontrol*
  This attribute operations in tandem with the *maxsize* attribute. It can be set to *auto*, *none*, *error* or *block*. *pause* and *wait* are aliases for *block*. The default setting is *auto*. In *error* mode, any attempt to add an object to a dataset which has already reached its maximum size raises an error. In *block* mode, the interpreter halts until the object count has decreased below the maximum size and then continue to move the object into the dataset. This mode is useful when the script is multi-threaded or the dataset operates a listener port for remote commands, because the number of objects in the dataset can change by these methods without involving the paused interpreter. The *none* mode disables the maximum size monitoring. Finally, the *auto* mode behaves like the *error* mode if there is only a single interpreter thread, and the dataset does not listen for remote commands, and like the *block* mode, if any of these two criteria are met.

- *swapthreshold*
  The maximum size of a dataset before ensemble and reaction objects in it are automatically swapped to disk, as they are by the explicit commands **`ens swapout`** or **`reaction swapout`**. The size check is performed at the moment new objects are added, and these new objects are the first to be swapped. The default value for this attribute can be set in the control array element*::cactvs(dataset_swap_threshold).* Its initial value is 10000. The default value for the embedded datasets in tables is controlled separately by ::*cactvs(table_swap_threshold),* which is also initially set to 10000.

  If this value is set to a negative value, all dataset elements which are currently swapped out are loaded back in. If it is set to a positive value, and the number of not currently swapped out objects of the dataset is more than the new limit, excess objects are swapped beginning from the end of the dataset queue until the in-memory object count of the dataset satisfies the new constraint. If the limit is increased, but not set to a negative unlimited value, the object swap status is not modified.

- *tablecount*
  A read-only attribute reporting the number of table objects currently contained in the dataset.

- *targeteod*
  The target value of the *eod* attribute. Once it matches or exceeds this value, the dataset is not expected to receive any more items. The initial value of this attribute is one.

- *threadcount*
  A read-only attribute returning the number of **Tcl** interpreter threads associated with the dataset. Normal datasets have no associated threads and return zero. This command is equivalent to the length of the list returned by the *threads* attribute, and the threads included in this count are the same.

- *threads*
  A read-only attribute returning the **Tcl** interpreter thread handles of the threads associated with the dataset (see **`dataset addthread`** command). Datasets without threads return an empty list. The handles are compatible with the standard **Tcl** thread package. Remote communication listener threads (see *port* attribute) are independent of **Tcl** support, do not have a **Tcl** handle, and are not listed by this command.

- *timeout*
  A timeout in seconds to use with the **`dataset wait`** command. A negative value means an infinite wait period, and zero no wait period. The default setting is minus one.

- *tooltip*
  If the toolkit was compiled with factory support, this is the tooltip of the object icon on a workbench. This attribute can be changed.

- *uuid*
  The automatically generated object instance **UUID**. This ID is independent of the **UUID** triple (class/instance/version) associated with the authorship attributes and intended for public dissemination. This attribute is read-only, unique for every dataset object - even duplicates -, and independent of its contents or pedigree.

- *version*
  A version number of the dataset. This is a string in a 1.2.3 (or shortened) style.

- *versionuuid*
  The version **UUID** associated with this dataset object as per its authorship attributes.
- *x*
  If the toolkit was compiled with factory support, this is the x coordinate of the object icon on its workbench. This attribute can be changed.
- *y*
  if the toolkit was compiled with factory support, this is the y coordinate of the object icon on its workbench.This attribute can be changed.

Examples:

```
dataset set $dhandle D_NAME "New lead structures"
dataset set $dhandle E_NAME "Lead (metal)"
```

The first line is a simple set operation for a dataset property. The second line shows how to set properties of multiple ensembles in one step. The same property value is assigned to all ensembles.

```
dataset set $dhandle port 10001 passphrase blockbuster
```

Set up a listener thread on port 10001 which accepts connections from remote interpreters which need to present the pass phrase as credential. Remote interpreters can add (**`ens move`**, **`reaction move`**, **`table move`**) or remove (**`dataset pop`**) objects to or from this dataset, as well as query the dataset object count (**`dataset count`**). Objects are transferred over the network connection as serialized objects to and from the remote interpreters.

dataset setparam

```
dataset setparam dhandle property ?key value?...
dataset setparam dhandle property dictionary
d.setparam(property,?key,value?...)
d.setparam(property,dict)
```

Set or update a property computation parameter in the parameter list of a valid property. This command is described in the section about retrieving property data.

The return value is the updated property computation parameter dictionary.

Example:

```
dataset setparam $dhandle D_GIF comment "Top Secret"
```

dataset show

```
dataset show dhandle propertylist ?filterset? ?parameterdict?
d.show(property=,?filters=?,?parameters=?)
Dataset.Show(items,property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **`dataset get`** command. The difference between **`dataset get`** and **`dataset show`** is that the latter does not attempt computation of property data, but raises an error if the data is not present and valid. For data already present, **`dataset get`** and **`dataset show`** are equivalent.

The **Python** class method is a one-shot command. The transient dataset created from the initialization items is automatically deleted when the command finishes.

dataset sort

```
dataset sort dhandle {property ?direction ?cmpflags ?cmpvalue???}...
d.sort((property,?direction,?cmpflags,?cmpvalue???),...)
```

Sort a dataset according to property values of the objects in the dataset. If no sort property set is specified, the default sort properties are `E_NATOMS` (number of atoms) and, for breaking ties, `E_WEIGHT` (molecular weight) and finally `E_HASHISY` (stereo isotope hash code).

Every sort item is interpreted as a nested list/tuple and can have from one to four elements. The first, mandatory element is the sort property, or one of the magic names *record* (or *#record*) or *random* (*#random)*. The next optional element is the sort direction, specified as *up* (or *ascending*) or *down* (*descending*). The default sorting order is ascending. The final optional comparison flags parameter can be set to a combination of any of the values allowed with the **`prop compare`** command. The default is an empty flag set. Properties in the sort list have precedence in the order they are specified in. Object property values of comparison list entries to the right in this list are only considered if the comparison of all data values of list elements to the left results in a tie.

If a comparison value is supplied as fourth argument, the sort utilizes the comparison results of dataset object property values against this value for ranking, not the direct comparison result between the dataset object property values. This is for example useful when sorting according to a bitvector similarity value to an external structure.

The magic property name *record* sorts by the object index in the dataset. Sorting upwards on this property does not change the object sequence in the dataset, and sorting downwards reverses it. This pseudo property is always added as a final implicit criterion, so that the sequence order of objects tied in all explicit comparisons is preserved. The other magic property name *random* assigns a random value to all dataset objects and sorts on this value, yielding a random object sequence.

The command returns a list of the handles of the objects controlled by the dataset in the newly sorted order. Simultaneously, the objects are physically moved within the dataset, so the sort has a persistent effect. The same result list may later be obtained by a **`dataset objects`** command.

It is possible to sort transient datasets, but this makes sense only if the object list sequence returned as command result is captured and used later, because the sort effect is not persistent since there exists no permanent dataset object.

Examples:

```
dataset sort $dhandle {E_NAME up {ignorecase lazy}]
```

The example sorts the dataset according to the compound name (property `E_NAME`, data type string) in alphabetic order, using a lazy (ignoring whitespace and punctuation) and case-insensitive comparison mode.

```
dataset sort $dhandle {E_NATOMS down} {E_NRINGS up}
```

Sort the dataset in such a way that the ensembles with the largest number of atoms, and among these those with the smallest number of rings, come first.

```
dataset sort $dhandle random
```

This command randomizes the object order in the dataset.

```
dataset sort $dhandle {*}$sortlist
```

This is the recommended construct when using a sort property list store in a **Tcl** variable as command argument. Older versions of the **`dataset sort`** command used a single sort argument parameter instead of a variable-size argument set.

dataset sqldget

```
dataset sqldget dhandle propertylist ?filterset? ?parameterdict?
d.sqldget(property=,?filters=?,?parameters=?)
Dataset.Sqldget(items,property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **`dataset get`** command. The differences between **`dataset get`** and **`dataset sqldget`** are that the latter does not attempt computation of property data, but initializes the property value to the default and returns that default, if the data is not present and valid; and that the **SQL** command variant formats the data as **SQL** values rather than for **Tcl** or **Python** script processing.

The **Python** class method is a one-shot command. The transient dataset created from the initialization items is automatically deleted when the command finishes.

dataset sqlget

```
dataset sqlget dhandle propertylist ?filterset? ?parameterdict?
d.sqlget(property=,?filters=?,?parameters=?)
Dataset.Sqlget(items,property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **`dataset get`** command. The difference between **`dataset get`** and **`dataset sqlget`** is that the **SQL** command variant formats the data as **SQL** values rather than for **Tcl** or **Python** script processing.

The **Python** class method is a one-shot command. The transient dataset created from the initialization items is automatically deleted when the command finishes.

dataset sqlnew

```
dataset sqlnew dhandle propertylist ?filterset? ?parameterdict?
d.sqlnew(property=,?filters=?,?parameters=?)
Dataset.Sqlnew(items,property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **`dataset get`** command. The differences between **`dataset get`** and **`dataset sqlnew`** are that the latter forces re-computation of the property data, and that the **SQL** command variant formats the data as **SQL** values rather than for **Tcl** or **Python** script processing.

The **Python** class method is a one-shot command. The transient dataset created from the initialization items is automatically deleted when the command finishes.

dataset sqlshow

```
dataset sqlshow dhandle propertylist ?filterset? ?parameterdict?
```

```
d.sqlshow(property=,?filters=?,?parameters=?)
Dataset.Sqlshow(items,property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **`dataset get`** command. The differences between **`dataset get`** and **`dataset sqlshow`** are that the latter does not attempt computation of property data, but raises an error if the data is not present and valid, and that the **SQL** command variant formats the data as **SQL** values rather than for **TCL** or **PYTHON** script processing.

The **PYTHON** class method is a one-shot command. The transient dataset created from the initialization items is automatically deleted when the command finishes.

dataset statistics

```
dataset statistics dhandle property
d.statistics(property)
```

Get basic statistics on the property values of the objects in the dataset. The property can be a basic property or a property field, but its element data type needs to be cast-able to a simple numeric type. In addition, it must be directly attached to any of the objects which can be members of a dataset, e.g. an ensemble property, but not an atom property.

If the property data is not present on an object, an attempt is made to compute it. In case that fails, or a dataset member object is not of a type matching the property, these objects are silently skipped.

The return value is a dictionary containing the number of objects in the dataset which were used for the statistics (key *n*), the sum of property values (*sum*), the property value average (*avg*) and the property data standard deviation (*stddev*). The latter three values are floating points, regardless of the property data type. In case any of these values are not computable, for example because there were an insufficient number of objects, the reported value is zero.

The command verb can be abbreviated as *stats*.

Example:

```
set d [dataset statistics $dh E_WEIGHT]
puts „Avg: [dict get $d avg]"
```

dataset subcommands

```
dataset subcommands
dir(Dataset)
```

Lists all subcommands of the **`dataset`** command. Note that this command does not require a dataset handle.

dataset tables

```
dataset tables dhandle ?filterset? ?filtermode? ?recursive?
d.tables(?filters=?,?mode=?,?recursive=?)
```

Return a list of all the handles or references of the tables in the dataset. Other objects in the dataset (ensembles, reactions, datasets, networks) are ignored. The object list may optionally be filtered by the filter list, and the result further modified by a standard filter mode argument.

If the *recursive* flag is set, and the dataset contains other datasets as objects, tables in these nested datasets are also listed.

Example:

```
set n [dataset tables $dhandle {} count]
```

dataset taint

```
dataset taint dhandle propertylist/changeset ?purge?
d.taint(property=,?purge=?)
```

Trigger a property data tainting event which acts on the dataset data, and all objects and their data contained in the dataset.

The parameters of this command are the same as for **`ens taint`** and explained there.

Example:

```
dataset taint $dhandle A_XYZ
```

All property data on the dataset and the dataset members is invalidated if it directly or indirectly depends on the 3D atomic coordinates.

The command returns the original object handle or reference.

dataset threadexec

```
dataset threadexec ?maxthreads? ?substitutiondict? scriptbody
```

Execute a script on the objects in the dataset in parallel in multiple threads. The number of threads is by default the lesser of 16 or the number of objects in the dataset, but this can be configured. If there are more dataset objects than threads, threads are started in a groupwise fashion. In the function body, standard **Tcl/Tk** percent substitution is performed. The default substitutions are **`%D`** for the dataset handle, and **`%o`** for the thread-specific dataset object. Other custom substitutions can be configured in the optional substitution dictionary, in a letter /value (no percent prefix) format.

There are some limitations on what the object threads can do. They are allowed to delete their own current object, or move it outside the dataset, but not other objects in the dataset. Additional objects may be appended to the dataset (they are not subject to processing by the original command), but not inserted in random positions. Computation in the script body must reach the end of the script, or be ended by **`return`** or **`break`** statements. An error in any of the threads stops the command. All threads of a group must have finished before a new group is started.

The command returns the dataset handle if the dataset is not virtual.

Because of multi-threading issues, there is no **Python** version of the command.

dataset transfer

```
dataset transfer dhandle propertylist ?targethandle? ?targetpropertylist?
d.transfer(properties=,?target=?,?targetproperties=?)
```

Copy property data from one dataset to another dataset or other major object, without going through an intermediate scripting language object representation, or alternatively dissociate property data from the dataset. If a property in the argument property list is not already valid on the source dataset, an attempt is made to compute it.

If a target object is specified, the return value is the handle or reference of the target object. The source dataset and the target object cannot be the same object.

If a target property list is given, the data from the source is stored as content of a different property on the target. For this, the data types of the properties must be compatible, and the object class of the target property that of the target object. No attempt is made to convert data of mismatched types. In case of multiple properties, the source property list and the target property list are stepped through in parallel. If there is no target property list, or it is shorter than the source list, unmatched entries are stored as original property values, and this implies that the object class of the source and target objects are the same.

If no target object is specified, or it is spelled as an empty string or **PYTHON** **`None`**, the visible effect of the command is the same as a simple **`dataset get`**, i.e. the result is the property data value or value list. The property data is then deleted from the source object. In case the data type of the deleted property was that of a major object (i.e. an ensemble, reaction, table, dataset or network), it is only unlinked from the source object, but not destroyed. This means that the object handles returned by the command can henceforth the used as independent objects. They can be deleted by a normal object deletion command, and are no longer managed by the source object.

Example:

```
dataset transfer $dh D_SVG_IMAGE $lh L_1DPATTERN_SVG_IMAGE
```

This command performs a data transfer between different object classes, with change of the property under which the content is stored.

dataset transform

```
dataset transform dhandle SMIRKSlist ?direction? ?reactionmode?
   ?selectionmode? ?flags? ?overlapmode? ?{?exclusionmode? excludesslist}?
   ?maxstructures? ?timeout? ?maxtransforms? ?niterations? ?statusvariable?
```

```
d.transform(transforms=,?direction=?,?reactionmode=?,?selectionmode=?,?flags=?,
   ?overlapmode=?,?excludess=?,?maxstructures=?,?timeout=?,?maxtransforms=?,
   ?iterations=?)
```

```
Dataset.Transform(items,transforms=,?direction=?,?reactionmode=?,
   ?selectionmode=?,?flags=?,?overlapmode=?,?excludess=?,?maxstructures=?,
   ?timeout=?,?maxtransforms=?,?iterations=?)
```

This command is complex, but very similar to the **`ens transform`** command. Please refer to that command for a full description of the command arguments.

The major difference of **`dataset transform`** is that the start structure set is not a single ensemble, but rather the set of all ensembles in the dataset. Any dataset items which are not ensembles are ignored. The return value is, just as with the **`ens transform`** command, a list of result ensembles. These do not become part of the input dataset.

Example:

```
dataset transform [ens get $ehandle E_KEKULESET] $trafolist bidirectional \
   multistep all {preservecharges checkaro setname}
```

This command first expands an ensemble object into a set of Kekulé structures. The property data type of the `E_KEKULESET` property is a dataset, so its handle is returned, and this dataset is then submitted for further transformation, which in this case involves manipulations of bonds in aromatic systems and thus is dependent on the Kekulé structures of the input ensembles.

The dataset variant of the transform command does not allow the use of marked or unmarked atom or bond specifications in the exclusion substructure list. Normal substructures are supported, and are applied to all start structures.

The **Python** class method is a one-shot command. The transient dataset created from the initialization items is automatically deleted when the command finishes.

dataset unique

```
dataset unique dhandle {property ?direction? ?cmpflags?}..
d.unique((property,?direction,?cmpflags,?cmpvalue???),...)
```

This command removes duplicate objects from the dataset and destroys them. Object equivalence is determined by pair-wise comparison of one or more properties. If all these properties are identical for any two objects, one of them is deleted. If no properties are specified, the default is the single property `E_HASHISY`, the standard isotope- and stereo-aware ensemble hash code.

The command returns labels or references of the ordered list of objects remaining in the dataset after deletion. The command is closely related to the **`dataset sort`** command, and the same restrictions on usable sort properties apply. Internally, the command performs a sort first, in order to avoid a quadratic growth of pair-wise comparisons. This has the side effect that the object order in the dataset is not preserved. Instead, the surviving objects are listed in ascending (by default) or descending (if the corresponding optional sort direction argument is set accordingly) values of the sort properties. The interpretation of the optional comparison flags and sort direction arguments, as well as the priority of the properties, and the special considerations when working on transient datasets, are the same as for the command **`dataset sort`**.

Example:

```
molfile read $fh $dh all
dataset unique $dh
```

This command first reads a complete file into a dataset, and then discard duplicates, using the default isotope- and stereo-aware structure hash code.

dataset unlock

```
dataset unlock dhandle propertylist/dataset/all
d.unlock(property=)
```

Unlock property data for the dataset object, meaning that they are again under the control of the standard data consistency manager.

The property data to unlock can be selected by providing a list of the following identifiers:

- Property names or references
  Valid property instances on the dataset object are unlocked. Non-existent data is silently ignored. It is not possible to unlock individual property fields.
- *all*
  All valid dataset object properties are unlocked.
- *dataset*
  This is an object class identifier. All property data which is controlled by the dataset major object and attached to the specified object class is unlocked. Since datasets do not incorporate minor objects, this identifier is equivalent to *all*.

Property data locks are obtained by the **`dataset lock`** command.

This command does not recurse into the objects contained in the dataset.

The return value is the original dataset handle or reference. If the argument was a transient dataset (only possible for **Tcl**), the result is an empty string.

dataset unpack

```
dataset unpack string ?compressionlib)
Dataset.Unpack(data=,?compressionlib=?)
```

Generate a dataset complete with all elements it contains from a packed, base64-encoded serialized object string, as it is generated by the complementary **`dataset pack`** command.

The return value is the handle or reference of the new dataset. All objects in the new dataset also are assigned standard handles, which can be retrieved with the usual commands such as **`dataset ens`** and **`dataset reactions`**.

The default compression library is *zlib*. For more options, see **`dataset pack`**.

Note that this command does not take a dataset handle as argument, but a pack string.

Example:

```
dataset unpack [dataset pack $dhandle]
```

This example is effectively the same as a **`dataset dup`** operation, but of course less efficient, because the objects have to be serialized, compressed, and base64-encoded and the same sequence of operations run backward again.

dataset valid

```
dataset valid dhandle propertylist
d.valid(property/propertysequence)
```

Returns a list of boolean values indicating whether values for the named properties are currently set for the dataset. No attempt at computation is made. For **Python**, where single-item lists are syntactically not the same as a single value, the return value is a single boolean if the argument was a string or a property reference, and only a single property was decoded.

Example:

```
dataset valid $dhandle D_NAME
```

reports whether the dataset is named (has a valid `D_NAME` property) or not.

`dataset has` is an alias to this command.

dataset verify

```
dataset verify dhandle property
d.verify(property)
```

Verify the values of the specified property on the dataset. The property data must be valid, and a dataset property. If the data can be found, it is checked against all constraints defined for the property, and, if such a function has been defined, is tested with the value verification function of the property.

If all tests are passed, the return value is boolean 1, 0 if the data could be found but fails a test, and an error condition otherwise.

dataset wait

```
dataset wait dhandle ?size|query? ?script?
d.wait(?query=?,?size=?,?function=?)
```

Suspend the interpreter until the number of objects in the dataset has reached a threshold, or an object which satisfies a query expression can be found. The syntax of query expressions is the same as in the **`dataset scan`** command. Query parsing is attempted if the argument is not a simple integer. If no explicit size or query expression is specified, or an empty string (or **`None`** for Python) is passed as this parameter, the command uses the value of the *highwatermark* dataset attribute as default value for an implicit size threshold condition.

Another dataset attribute which has an influence on the execution of the command is the *timeout* attribute. If the dataset size has not grown to the required size, or no object which satisfies the query expression was added to the dataset after waiting for the *timeout* number of seconds, an error is raised. By default, the maximum wait period is indefinite, which corresponds to a negative timeout value. If the timeout value is set to zero, the wait condition must be met immediately, or an error results. However, no error is raised if the *eod/targeteod* dataset parameter pair indicates that no more data can be expected to be added in the dataset. In that case, the result is an empty string, or **`None`** for Python.

If no script function parameter is used, the return value of the command is the number of objects the dataset holds in case of an explicit or implicit size condition, or the handle/reference of the first matching object in case of a query expression.

If the object count already exceeds the threshold, or a matching object can be found at the moment the command is executed, the command returns immediately.

In the **Tcl** case, and in the presence of a script body parameter, the script is executed whenever the wait condition is met. If the script is ended with a *continue* statement, or simply reaches the end of the code block, the wait loop is automatically restarted. If the script reports an error, or is left via a *break* or *return* statement, the loop is terminated.

For **Python**, instead of the script body, a function name or reference can be used. This function is called in local scope with a single argument, which is either the current dataset item count in case of a simple threshold condition, or the reference of the object matching the query expression. Within the **Python** functions, the normal *break* and *continue* loop control commands cannot be used to to scope limitations. Instead, the custom exceptions *BreakLoop* and *ContinueLoop* can be raised. These are automatically caught and processed in the loop body handler code.

This command is mostly useful when running multi-threaded scripts, or when the dataset has an active remote command listener on a port. Under these circumstances, new objects may arrive in the dataset without participation of the local, waiting and stopped interpreter, which can then be processed.

While a **`dataset wait`** command is pending, the dataset cannot be deleted. Since it is possible that other threads or port monitors further update the dataset between the time the wait condition is met and script processing commences, action scripts should be prepared to see more or less items in the dataset than there were immediately after the trigger event.

Example:

```
loop n 1 $nrecs {
   set eh [dataset wait $dh “E_FILE(startrec) = $n”]
   molfile write $fh $eh
   ens delete $eh
}
```

This is a part of a simple write thread which writes back processed ensembles in the same order as they were read from an input file. In case there are multiple processing threads, it is likely to happen that the computation on an ensemble read from a higher input file record finishes before another with a smaller record number and thus the sequence of the ensembles to be written as delivered in the output queue becomes out of sync. By waiting for ensembles in the input record sequence the original order is preserved. More robust versions of such a script should handle the case of ensembles from a specific input record never appearing in the dataset and similar sources of disruption.

dataset weed

```
dataset weed dhandle keywords
d.weed(keywordsequence)
d.weed(?keyword?,...)
```

This command performs standard clean-up operations on all ensembles and reactions in the dataset. The supported operations are described in more detail in the section on the equivalent **`ens weed`** command.

The return value of this command is the dataset handle or reference.

dataset xlabel

```
dataset xlabel dhandle propertylist ?filterset? ?filterprocs?
d.xlabel(property=,?filters=?,?filterfunctions=?)
```

This command is rather complex and closely related to the **`dataset extract`** command. Its purpose is to extract handle/reference and label information for selected subsets of the dataset. The return value is a nested list. The sublists consist of the object handle or reference, the object label (if the object does not have a label, 1 is substituted), and the dataset object index. The dataset object index starts with zero.

The selection of the class of objects which are extracted is performed indirectly via the property list. For practical purposes, this list should be a single property. Its object association type determines the class of objects selected. For example, `A_LABEL` or `A_SYMBOL` returns atom labels, while `B_ORDER` returns bond labels and `E_NAME` select complete ensembles, with 1 as pseudo ensemble label.

The objects for which data is returned can further be filtered by a standard filter set, and additionally by a list of filter procedures (for **Tcl**, specified as procedure names) or functions (for **Python**, specified as function names or function references). These procedures or functions are called with the respective object handles/references and object labels as arguments. For example, a callback function used in an atom retrieval context would be called for each atom with its ensemble handle or reference and the atom label as arguments. If major objects without a label are checked, such as complete ensembles, 1 is passed as the label. The callback procedures are expected to return a boolean value. If it is *false* or 0, the object is not added to the returned list, and the other check procedures are no longer called.

The command currently only works on ensembles in the dataset, ignoring any reactions, tables, datasets or networks which may be present.

This command is primarily useful for the display of filtered minor object data from datasets, such as atom property values for specific types of atoms.

Example:

```
set dhandle [dataset create [ens create O] [ens create C=C]]
dataset xlabel $dhandle A_LABEL !hydrogen
dataset xlabel $dhandle B_ORDER doublebond
```

First, a dataset with two ensembles (*water* and *ethene*) is created. This dataset is then queried. The first query is for all atoms in it which are not hydrogen. The returned list is

```
{ens0 1 0} {ens1 1 1} {ens1 2 1}
```

In object *ens0*, which is the first object in the dataset, atom *1* passes the filter. In object *ens1*, which is the second object in the dataset, atoms with label *1* and *2* pass. The second query asks for the labels of double bonds in the dataset. The use of property `B_ORDER` is arbitrary - any other bond property would do as well. The return value of this command is

```
{ens1 1 1}
```

which indicates that only the bond with label 1 in object *ens1*, which is the second object in the dataset, fulfills this condition.

The *ens* Command

The ens command is the generic command used to manipulate molecular ensembles. Ensembles are the most commonly used chemistry major object. Ensembles contain atom, bonds, molecules and other minor objects.

The syntax of this command follows the standard schema of *command/subcommand/majorhandle*. Since molecular ensembles are major objects, they are not addressed via labels.

Similar to the functionality of *molfile* and dataset objects, ensembles can be persistent, or transient. Persistent ensembles are those created by the **ens create** command or similar functions. They possess a handle and exist until explicitly deleted. Transient ensembles only exist for the duration of a single command. They are deleted as soon as the command finishes, regardless whether the command was successful or not.

Examples:

```
ens get $ehandle E_SMILES
ens merge [ens create CCC] [ens create CCC]
ens get lycorine E_CID
```

This is the list of officially supported subcommands:

ens add

```
ens add ehandle ?ehandle_list?...
e.add(?eref/erefsequence?,...)
e += eref
```

This command performs the same operation as the **ens merge** command, but preserves the ensembles in the merge lists (argument four and onwards in the **Tcl** command variant). The base ensemble (third argument) is modified.

Please refer to the **ens merge** command for a more detailed documentation.

The **Python** arithmetic command returns a reference of the original ensemble, not the new first atom label or reference of the merged ensemble (see again **ens merge**).

ens align3d

```
ens align3d ehandle box/center/masscenter/pmi ?usehydrogens? ?property?
e.align3d(?mode=?,?usehydrogens=?,?coordinateproperty=?)
```

Perform a 3D alignment by modifying standard atom coordinates property `A_XYZ`, or an alternative explicitly specified atomic coordinate property.

The possible alignment modes are

- *box*
  move center of enclosing 3D coordinate box to origin
- *center*
  move average atom coordinates to origin
- *masscenter*
  move mass-weighted atom coordinates to origin

- *pmi*
  align ensemble to principle moments of inertia (largest on x axis), and move the mass-weighted center to the origin.

By default all atoms are used to compute the alignment rotation and movement vectors, including hydrogens. If these should be omitted from computing the movement vectors (but not the subsequent atom movement), the optional *usehydrogens* parameter can be set to *false*.

The command returns the handle or reference of the ensemble.

ens append

```
ens append ehandle ?property value?...
e.append({?property:value,?...})
e.append(?property,value,?...)
```

Standard data manipulation command for appending property data. It is explained in more detail in the section about setting property data.

The command returns the first data value.

Example:

```
ens append $ehandle E_NAME "_linker"
```

ens assign

```
ens assign ehandle srcproperty dstproperty
e.assign(srcproperty=,dstproperty=)
```

Assign property data to another property on the same ensemble. Both properties must be associated with the ensemble object class. This process is more efficient than going through a pair of **ens get/ens set** commands, because in most cases no string or TCL/PYTHON script object representations of the property data need to be created.

Both source and destination properties may be addressed with field specifications. A data conversion path must exist between the data types of the involved properties. If any data conversion fails, the command fails. For example, it is possible to assign a string property to a numeric property - but only if all property values can be successfully converted to that numeric type. The reverse example case always succeeds, out-of-memory errors and similar global events excluded.

The original property data remains valid. The command variant **ens rename** directly exchanges the property name without any data duplication or conversion, if that is possible. In any case, the original property data is no longer present after the execution of this command variant.

The command returns the original object handle for TCL, or object reference for PYTHON.

Examples:

```
ens assign $ehandle A_XY A_XY%
ens assign $ehandle E_NMRSPECTRUM(spectrometer) E_METHOD
ens rename $ehandle E_IDENT E_NAME
```

ens atoms

```
ens atoms ehandle ?filterset? ?filtermode?
e.atoms(?filters=?,?mode=?)
```

Standard cross-referencing command to obtain the labels or references of the atoms the ensemble contains as minor objects. This is explained in more detail in the section about object cross-references.

Examples:

```
ens atoms $ehandle
ens atoms $ehandle hydrogen
ens atoms $ehandle !hydrogen count
```

The first example simply returns a list of the labels of the atoms the ensemble contains as minor objects. The second example returns the atom label(s) of all hydrogen atoms in the ensemble. If there are no such atoms, an empty list is returned. The final example counts the number of non-hydrogen atoms in the ensemble.

ens bondangles

```
ens bondangles ehandle ?filterset? ?filtermode?
e.bondangles(?filters=?,?mode=?)
```

Standard cross-referencing command to obtain the labels or references of the bond angle objects the ensemble contains as minor objects. This is explained in more detail in the section about object cross-references.

ens bonds

```
ens bonds ehandle ?filterset? ?filtermode?
e.bonds(?filters=?,?mode=?)
```

Standard cross-referencing command to obtain the labels or references of the bonds the ensemble contains as minor objects. This is explained in more detail in the section about object cross-references.

Examples:

```
ens bonds $ehandle
ens bonds $ehandle doublebond
ens bonds $ehandle carbon count
```

The first example simply returns a list of the labels of the bonds the ensemble contains as minor objects. The second example returns the bonds label(s) of all double bonds in the ensemble. If there are no such bonds, an empty list is returned. The final example counts the number of bonds which involve one or more carbon atoms in the ensemble.

ens cast

```
ens cast ehandle dataset/ens/reaction/table ?propertylist?
e.cast(objectclass=,?properties=?)
```

Transform the ensemble into a different object. Depending on the target object class, the result is as follows:

- *dataset*
  A new dataset which contains which contains the ensemble as first and only object.
- *ens*
  Only supplied for the sake of completeness. This mode does nothing.

- *reaction*
  A new reaction, which contains the original and a duplicate of the ensemble as reagent and product components, and an auto-generated 1:1 `A_MAPPING` property.
- *table*
  A new table with one row and automatically generated columns for all properties of the input ensemble of the *ens* (E_*) object class. The row is filled with the input ensemble data, and the ensemble is moved to the internal dataset of the table.

If the optional property list is specified, an attempt is made to compute the listed properties before the cast operation, so that they may become a part of the new object. No error is raised if a computation fails.

The command returns the handle (reference for **Python**) of the new object, or the input object in case of mode *ens*.

ens clear

```
ens clear ehandle ?keepensprops?
e.clear(?keepensproperties=?)
```

This command resets an ensemble to a virgin state. All minor objects and all property data of the ensemble are deleted. However, the ensemble handle or reference remains valid, representing an ensemble without any atoms, bonds, rings or other minor objects. If the optional argument is set to a true value, ensemble-class properties (`E_*`) are not deleted, but everything else still is.

Ensemble membership in datasets, reactions, etc. is not changed by this command.

The command returns the original handle or reference.

ens compare

```
ens compare ehandle ehandle2
e.compare(eref/ehandle)
```

Compare two ensembles, yielding a stable sort order. The compared attributes are, in this order, the number of atoms, the number of bonds, the ensemble molecular weight, the number of ESSSR rings and finally the stereo- and isotope aware 64-bit hashcode (`E_ISOTOPE_STEREO_HASHY`). The command returns 1 if the first ensemble is larger, -1 if the second is larger, and 0 if they are identical according to the comparison scheme.

The compared property values, with the exception of the final hashcode tiebreaker, are compatible with the **RDKit** model.

ens copy

```
ens copy src_ehandle dst_ehandle
e.copy(eref_dst)
```

Create a copy of the input ensemble into the framework of an existing ensemble. The old data of the destination ensemble is destroyed, but its handle or reference is reused for the copy. The destination handle can be an empty string, **`#new`**, **`#auto`** or **`None`** for **Python**. In that case, the ensemble is duplicated and a new handle assigned.

This command is useful when an ensemble handle or reference is potentially stored in unknown locations and the ensemble data needs to be updated.

The return value of the command is the handle or reference of the destination ensemble. It is allowed to copy an ensemble onto itself.

Example:

```
set eh1 [ens create CC]
set eh2 [ens create CCC]
ens copy $eh1 $eh2
```

After the example code sequence, both ensembles represent ethane, the first compound. However, these are independent ensembles. Any further modifications of the ensemble data on any of the ensembles will not be seen by the other.

The command returns the handle or reference of the target ensemble.

ens create

```
ens create ?codestring? ?mode? ?datasethandle? ?macroset?
Ens(?data=?,?mode=?,?dataset=?,?macroset=?)
Ens.Create(?data=?,?mode=?,?dataset=?,?macroset=?)
```

This command creates a new molecular ensemble and returns its handle or reference. If none of the optional arguments are specified, or the argument string is an empty string (or `None` for **PYTHON**), an empty ensemble without any atoms or bonds is created. These may later be populated with commands like *atom create*.

If data string may either begin with an automatically recognized prefix, or an automatic format detection process is initiated. Recognized prefixes are:

- aa:
  Decode a 1-letter or 3-letter case-sensitive amino acid sequence. Stereochemistry is assumed to be natural (i.e. L amino acids). The first amino acid has the free amino group, the last the free carboxyl group.

- aldrich:
  Decode a Sigma-Aldrich catalog number via the *sigmaaldrich.com* Website. There are a couple of alias prefixes: sigma:, sial: milliporesigma:, sigmaaldrich: and ms:

- cas:
  Decode **CAS** number via the **NCI** resolver, **PUBCHEM** or `commonchemistry.org`. Since properly formed **CAS** numbers are distinctive, they are also recognized without a prefix.

- cdx:
  Decode base64-encoded ChemDraw **CDX** data. This is the format sent to Web servers by the ChemDraw browser plug-in.

- chebi:
  Decode **CHEBI** ID

- chembl:
  Decode **CHEMBL** ID

- chemspider:
  Decode **CHEMSPIDER** ID
- cid:
  Decode **PUBCHEM CID**
- drugbank:
  Decode **DRUGBANK** ID
- emol:
- Decode **EMOLECULES** ID. *emolecules:* is an equivalent prefix.
- formula:
  Decode as formula, i.e. create elemental atoms, but no bonds
- inchi:
  Decode an **INCHI** string. Usually this is not a needed prefix since the standard beginning of an **INCHI** string (*InChI=)* is sufficiently unique to prevent misinterpretation. The prefix can be useful in case it is not known whether the InChI string has a proper lead-in. If the *InChI=* part has been stripped, the decoder does not automatically recognize the encoding. With the explicit prefix, InChI strings with and without the lead-in are decodable.
- jme:
  Decode as data string of **JME** Java structure editor
- kegg:
  Decode **KEGG** ID
- lincs:
  Decode a **LINCS** ID.
- mcule:
  Decode **MCULE** ID
- mesh:
  Decode **NCBI** Mesh ID
- mfcd:
  Decode MDL structure ID. The value following the colon can be either a simple number, or start with the *MFCD* prefix in upper case.
- name:
  Perform name resolution using the NCI resolver, OPSIN, KEGG or ChemSpider, depending on the system configuration. By default, first the NCI resolver and, if that fails, OPSIN are contacted.
- patran:
  Decode a **LHASA** 1D **PATRAN** query pattern. 2D **PATRAN** patterns can be decoded with `reaction create`.
- pdb:
  Decode a **PDB** ID (4 characters, initial number plus 3 alpha characters)

- querysln:
  Decode as Query**SLN** string.
- sid:
  Decode **PubChem SID**
- sln:
  Decode as **SLN** string
- smarts:
  Decode as **SMARTS** (explicitly not as **SMILES**)
- smiles:
  Decode as **SMILES** (explicitly not as **SMARTS**)
- strictsmiles:
  Decode as **SMILES** (explicitly not as **SMARTS**), and also use hypervalent hydrogen addition as per the original Daylight definition (see also the description of the `::cactvs(smiles_hypervalent_hydrogen_addition)` control variable).
- unii:
  Decode as **FDA UNII** code. Properly formed **UNII**s are also automatically recognized without a prefix.
- zinc:
  Decode **ZINC** ID
- quoted with ' or "
  Handled the same way as the *name:* prefix. These must be explicit quotes that are part of the string, not string syntax elements of the script. Example: `ens create "aspirin"` vs. `ens create \"aspirin\"` or `ens create 'aspirin'` - the latter two commands work as expected, the first does not, because the quotes are not an actual part of the string, and *aspirin* can be decoded (in a very lenient fashion) as **SMILES**, which has precedence.

The colon in the prefix may be omitted (except for the *name:* item), but this is not recommended, since it may lead to misinterpretation of the data if the prefix is also part of a valid structure encoding.

In addition, **URL**s as structure data argument are automatically detected and handled specially. If the **URL** is a data **URI**, it is unpacked and its payload processed in a second cycle. If it is an **HTTP** or **FTP URL**, the file is downloaded and its contents read a a structure file with automatic format detection. This is not identical to data **URI** processing: Data **URI**s are again interpreted as command arguments with all prefix and line notation interpretation, while file contents are only interpreted as a record in a structure data file.

If none of the above special cases are recognized, automatic interpretation is performed next. Currently, the encoding then may either be

- a **SMILES/SMARTS** string (see below on how to distinguish these)
- a hex-encoded **SMILES/SMARTS** string, as used by some Daylight tools
- an **InChI** string, with a proper lead-in (*InChI=*)

- a **CACTVS** packed serialized object string, as it is generated by the `ens pack` command
- a **CACTVS** *Minimol* object in binary or *base64*-encoded form
- a plain text or *base64*-encoded blob of the contents of a structure file record, such as an **MDL** SDfile. The format must be identifiable by the currently loaded set of structure file I/O modules. Since the data has no file name, automatic loading of modules is not possible.
- a **PUBCHEM CID** - any simple integer argument is interpreted as **CID**
- a **CAS** number which is looked up on the Internet provided general Internet access is enabled in the toolkit
- an **MDL** structure ID, starting with a proper lead-in (*MFCD*), followed by eight digits, which is also resolved by Internet access to the *chemsynthesis.com* site if possible.
- an **INCHI** key, with or without lead-in (*InChIKey*=). This only works for keys which can be looked up via the NCI resolver over the Internet.
- a properly formed **FDA UNII** code.
- a structure file record image as produced by the **MYSQL** database `compress()` function (i.e. 4 byte binary uncompressed size prefix plus *zlib*-compressed content). This is primarily useful when the command is used in the context of the **MYSQL** database cartridge.
- a compound name as last resort, which is by default looked up via the **NCI** resolver and the **OPSIN** service

In the absence of a prefix, the encoding is automatically detected. With the exception of **PUBCHEM** CIDs, the long form of a database ID must be used, not its simple integer value (i.e. a simple 70 is interpreted as **PUBCHEM** CID, while *CHEMBL70* or *chembl:70* are decoded as **CHEMBL** database IDs).

For the *base64*-encoded compressed records, the compression algorithm may be raw *zlib*, *gzip* or *zip* and its type is automatically detected.

In case one of the **SMILES**-class encoding schemes is used, the *mode* argument of the `ens create` command provides finer control of the decoding. By default, or when this argument is an empty string, the string is interpreted as standard **SMILES**, except when there are elements in the string which cannot occur in **SMILES** but in **SMARTS**. In **SMILES** mode, query expressions are only recognized to a very limited degree, and implicit hydrogens are automatically added. This decoding scheme may also be explicitly selected by specifying *hadd* as mode.

In order to force a full hydrogen addition to the raw decoded structure even if it would not be done otherwise, use the mode *forcehadd*.

Mode *strictsmiles* decodes **SMILES** with hydrogen addition but as if the *strictsmiles:* prefix was set. This is described above.

Mode *nohadd* is essentially the same as basic **SMILES** decoding, but implicit hydrogen addition does not happen. In any case, explicitly encoded hydrogen is decoded and preserved.

Mode *smarts* (or *query*) also skips hydrogen addition, but in addition the decoder now fully parses `SMARTS`, including `Recursive SMARTS,` but it also becomes less lenient in the area of superatom encodings and similar gray areas, in order to avoid ambiguity. The recognized `SMILES` dialect may be switched via the control variable *::cactvs(smiles_version).* The default is Daylight release 4.9 with **CACTVS** and **ELILILLY** extensions.

Mode *sln* forces the interpretation of the input string as **Sybyl Line Notation**. If the **SLN** I/O module has already been loaded, interpretation as **SLN** is automatically attempted in any case, but only after **SMILES** decoding has failed. Since there are strings which are both valid **SMILES** and **SLN**, but mean something different, this automatism can lead to misinterpretation, so if you know you are dealing with **SLN**, it is a good idea to specify it. The *sln* mode attempts to auto-load the **SLN** I/O module if it is not yet loaded. In case it cannot be loaded, this mode raises an error. Mode *querysln* is similar, but assumes the input is query**SLN**, not plain **SLN**.

The *3D* decoder mode prefers resolution of identifiers as 3D model instead of 2D connectivity. This has an effect only with a few select combination of identifiers and resolvers and should be considered experimental.

Instead of using an explicit decoder mode or a data prefix, it is also possible to supply the name of a property the structure data is an instance of. Examples are `E_SDF_STRING` or `E_SMILES`. Such properties are expected to provide suitable default decoder configuration data in their *fileformat* and *fileflags* attributes, and these are then used to decode the structure.

In *nohadd* decoder mode, the structure code is finally, if everything else fails, interpreted as a plain molecular formula. If the string is parsed successfully as a formula, a collection of atoms of the specified elements is created, without any bonds.

By default, or if the optional target dataset parameter is an empty string, the new ensemble is not a member of any dataset. It may be directly made a dataset member if a dataset handle is specified.

If a macro set name is specified, **SMILES** and **SMARTS** with macro definitions can be processed. Any patterns names which belong to the specified set are expanded. Set names, pattern names and expansion fragments are specified in the system macro table. Macro expansion is not available if the toolkit was compiled without table support.

Examples:

```
set eh [ens create]
set eh [ens create CCC]
set sshandle [ens create {[CH3][Cl,Br,I]} smarts]
set eh [ens create [decode -url C%23C] nohadd]
```

In case a structure is encoded as a string in a format which cannot be directly decoded by the **ens create** command (such as a plain string representation of an **MDL** molfile), the standard method is to load the appropriate file format decoder (if not built in, this is needed so that automatic format detection of the memory image record works), open the structure string as a memory-based structure file, and read from this file. This technique allows the input of multiple records from the in-memory file and thus is also useful in cases like a multi-record **SMILES** file encoded as a string.

Example:

```
filex load cdx
set fh [molfile open [decode -base 64 $cdxstring] s]
set eh [molfile read $fh]
molfile close $fh
```

ens dataset

```
ens dataset ehandle ?filterlist?
e.dataset(?filters=?)
```

Return the dataset handle or reference of the dataset the ensemble is part of. It the ensemble is not member of a dataset, or does not pass all of the optional filters, an empty string or **`None`** for **PYTHON** is returned.

Example:

```
ens dataset $ehandle
```

ens defined

```
ens defined ehandle property
e.defined(property)
```

This command checks whether a property is defined for the ensemble. This is explained in more detail in the section about property validity checking. Note that this is *not* a check for the presence of property data! The **`ens valid`** command is used for this purpose.

The command returns a boolean result.

ens delete

```
ens delete all
ens delete ?ehandlelist?...
e.delete()
Ens.Delete("all")
Ens.Delete(?erefsequence/eref/ehandle?,...)
```

Delete ensembles and the minor objects which are part of the deleted ensembles. The special parameter *all* may be used to delete all ensembles currently registered in the application, including those which are part of reactions or other major objects. Alternatively, any number of lists of ensemble handles may be specified for specific deletions.

The command returns the number of deleted ensembles.

For historic reasons, the same command may also be invoked as **`ens destroy`**.

Example:

```
ens delete $ehandle
ens delete $ehandlelist1 $ehandlelist2
```

ens dget

```
ens dget ehandle propertylist ?filterset? ?parameterdict?
e.dget(property=,?filters=?,?parameters=?)
Ens.Dget(data,property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **`ens get`** command. The difference between **`ens get`** and **`ens dget`** is that the latter does not attempt computation of property data, but rather initializes the property values to the default and return that default if the data is not yet available. For data already present, **`ens get`** and **`ens dget`** are equivalent.

The **Python** class method is a one-shot command. The transient ensemble created from the initialization items is automatically deleted when the command finishes. The data for the creation of the temporary ensemble is equivalent to the first argument of the standard constructor. Additional constructor parameters cannot be used.

ens dup

```
ens dup ehandle ?datasethandle? ?position? ?filterset? ?ctonlyflag?
e.dup(?dataset=?,?position=?,?filters=?,?ctonly=?)
```

Duplicate an ensemble. The return value is the handle or reference of the new ensemble.

The duplicate ensemble is placed into the same dataset as the source, if it is a member of a dataset. Specifying an explicitly empty dataset argument (including **None** for **Python**) places the duplicate outside any dataset, regardless of the dataset membership of the source ensemble.

If the duplicate is moved to a dataset, it is appended to the dataset end by default. This happens also if the position parameter is explicitly specified as *end* or an empty string. Otherwise, the ensemble is inserted at the given position, starting with 0. If the requested position is larger than the current size of the dataset, the ensemble is appended.

The filter parameter allows the selection of only a subset of atoms to be copied. All atoms which do not pass the filters are discarded, as are all bonds which connect to discarded atoms. If no atoms pass the filters, the result is an empty ensemble. By default, no atom filtering takes place, and all atoms and bonds of the original ensemble are part of the duplicate.

The final optional parameter can be used to make the duplicate lightweight. If this boolean parameter is set, the duplicate is limited to the basic connectivity information with all atom and bond properties, but it has no copies of properties of other object classes, and no copies of rings, molecules, groups or other minor object classes.

The **ens hdup** command is a variant of this command. It automatically adds a hydrogen set to the duplicate.

Examples:

```
ens dup $ehandle
ens dup $ehandle [dataset create] end ringatom
```

The first sample line is a standard use. The second example moves the duplicate into a newly created dataset, and isolates the ring systems. All other atoms are stripped.

ens exists

```
ens exists ehandle ?filterset?
e.exists(?filters=?)
Ens.Exists(eref=,?filters=?)
```

Check whether an ensemble handle or reference is valid. The command returns boolean 0 or 1. Optionally, the ensemble may be filtered by a standard filter list and it is reported as not valid if it does not pass the filters. If filters in the filter list operate on atom, bonds, or other minor objects, it is sufficient if a single minor object of the ensemble passes the filter.

Example:

```
ens exists $ehandle chlorine
```

Check whether the ensemble with the handle in variable **`$ehandle`** exists and, if it exists, whether it contains one or more chlorine atoms.

ens expand

```
ens expand ehandle ?allowambiguous? ?noimplicith?
e.expand(?allowambiguous=?,?noimplicith=?)
```

This command expands all superatoms in the ensemble. The mechanisms for the expansion of superatoms are described in detail for the **`atom expand`** command. This command is functionally equivalent, working on all atoms in the ensemble instead a single atom.

Example:

```
ens expand $ehandle
```

The command returns the total number of successfully expanded atoms.

ens expr

```
ens expr ehandle expression
e.expr(expression)
```

Compute a standard **SQL**-style property expression for the ensemble. This is explained in detail in the chapter on property expressions.

ens fill

```
ens fill ehandle ?property value?...
e.fill({?property:value,...})
e.fill(?property,value?,...)
```

Standard data manipulation command for setting data, ignoring possible mismatches between the lengths of the lists of objects associated with the property and the value list. It is explained in more detail in the section about setting property data.

Example:

```
ens fill $ehandle B_COLOR red
```

sets the color of the first bond in the ensemble to *red*.

ens filter

```
ens filter ehandle filterlist
e.filter(filters)
```

Check whether the ensemble passes a filter list. The return value is boolean 1 for success and 0 for failure.

Example:

```
ens filter [ens create CCCl] chlorine
```

checks whether the ensemble contains one or more chlorine atoms. If the filter operates on minor objects of the ensemble, it is sufficient to have a single ensemble minor object pass the filter condition.

ens forget

```
ens forget ehandle ?objclass?
e.forget(?objectclass=?)
```

Delete specific classes of minor objects and their data from the ensemble data structure. If no object class is specified, all minor object classes except atoms and bonds and the ensemble data are purged.

If the object class *ens* is specified, all property data attached to the ensemble object class (usually those properties starting with `E_*`) are deleted, but not the ensemble itself.

The command returns the original ensemble handle or reference.

ens formulamatch

```
ens formulamatch ehandle formula_expression ?other_elements?
e.formulamatch(query=,?other_elements=?)
```

Match the ensemble against a formula expression. Its syntax is the same as in formula queries in **`molfile scan`** and other scan commands.

There are several methods to specify whether any elements not mentioned in the formula expression may or must be present. If the *other_elements* flag is used, it has the highest priority. If may be set to 0 (no other elements allowed), 1 (allowed) or 2 (required), and if it is set, any prefix in the formula expression is ignored. If it is not used, a prefix in the formula expression may be used to control the matching. Supported prefixes are = (no other elements), >= (other elements allowed) and > (required). If no prefix is used, the default mode is an exact match without other elements.

The return value is the boolean match result.

Example:

```
ens formulamatch $eh >=C6
```

Matches any ensemble with has six carbon atoms.

```
ens formulamatch $eh C5-6(Cl+Br+I)2- 1
```

Matches an ensemble with five or six carbon atoms, two ore more heavy halogens, and potentially any other elements.

ens fragment

```
ens fragment ehandle atomlist ?datasethandle? ?position?
e.fragment(atomsequence=,?dataset=?,?position=?)
```

Create a new ensemble from a set of atoms in another ensemble. All bonds existing between those atoms are also preserved. The atoms can be selected with any standard atom selection syntax, with one selector per list element. Duplicate atom specifications are ignored. Atom specifications which cannot be resolved generate an error.

By default, the new ensemble becomes a member of the same dataset (if any) as the source ensemble, but this can be changed with the optional fifth argument. If no explicit position is given, the ensemble is appended to rear of the target dataset. The new ensemble only inherits the selected atoms and bonds plus stable atom and bond properties, but not other minor objects or ensemble data.

The command returns the handle or reference of the new ensemble object.

Example:

```
match ss $substructure $eh amap
set ehfrag [ens fragment $ehandle [unzip $amap 1]]
```

Above code sequence matches a substructure, and then extracts the matched structure part as a new ensemble.

ens get

```
ens get ehandle propertylist ?filterset? ?parameterdict?
ens get ehandle attribute
e.get(property=,?filters=?,?parameters=?)
e.get(attribute)
e[property/attribute]
e.property/attribute
Ens.Get(data,property=,?filters=?,?parameters=?)
Ens.Get(data,attribute)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

Examples:

```
ens get $ehandle {M_WEIGHT A_ELEMENT}
```

yields a nested list with two elements. The first element is a list of the molecular weights of all molecules in the ensemble. The second element is a list of the element numbers of all atoms in the ensemble. If the information is not yet available, an attempt is made to compute it. If the computation fails, an error results.

```
ens get $ehandle B_ORDER ringbond
```

gives the bond orders of all bonds of the ensemble which are ring bonds.

The format of the optional parameter list argument is a series of keyword/value pairs, as produced by the **Tcl** command *array get* or the standard **Tcl** dictionary commands. If a this parameter list is present as argument, and the requested property data is already valid for the ensemble, a check if made if all the specified parameters are the same as the parameters the present property data was computed with. If this is the case, the values are directly returned as usual. Otherwise, the data is discarded and re-computed.

If computation of the property data is performed, either because the parameter set was not matched, or the requested data was not valid, the computation integrates the specified parameter set into the parameters of the computation function. Parameters from the list *temporarily* override the global settings of these parameters in the property definition. Parameters used by the property computation function but not listed in the local parameter list are neither used for data validity checking, nor their value changed during the computation request. After the computation finishes, the old global parameter settings of the property definition are restored.

The use of a parameter list argument is primarily useful only if a single property is requested with this command, but its use with a multiple-property request is not illegal - the parameter list is simply applied to all properties in sequence.

The **Python** class method is a one-shot command. The transient ensemble created from the initialization items is automatically deleted when the command finishes. The data for the creation of the temporary ensemble is equivalent to the first argument of the standard constructor. Additional constructor parameters cannot be used.

Example:

```
ens get $ehandle E_GIF {} [dict create width 200 height 200 bgcolor white]
```

Variants of the **`ens get`** command are **`ens new`**, **`ens dget`**, **`ens jget`**, **`ens jnew`**, **`ens jshow`**, **`ens nget`**, **`ens show`**, **`ens sqldget`**, **`ens sqlget`**, **`ens sqlnew`**, and **`ens sqlshow`**.

Further examples:

```
ens get $ehandle E_NAME
ens get $ehandle A_FLAGS(boxed)
```

In addition to property data, the ensemble object possesses a few attributes, which can be retrieved with the ens **`get`** command (but not by its related sister subcommands like **`ens dget`**, **`ens sqlget`**, etc.). Some of them are also modifiable via **`ens set`**. These attributes are:

- *coords*
  If the toolkit was compiled with factory support, these are the coordinates of the object icon on its workbench, encoded as integer pair. This attribute can be changed.
- *deletable*
  Flag indicating whether this object can be deleted with a standard **`ens delete`** command. This attribute is read-only. Objects which are, for example, property data values or a part of a **`molfile loop`** command cannot be deleted by standard means.
- *failures*
  If the property computation failure cache is active, return a list of all properties which have failed computation for this ensemble after the last structural change. This attribute is read-only.
- *footer*
  If the toolkit was compiled with factory support, this is the footer of the object icon on a workbench. This attribute can be changed.
- *gflags*
  If the toolkit was compiled with factory support, this is the currently set object icon rendering flag collection.
- *header*
  f the toolkit was compiled with factory support, this is the header of the object icon on a workbench. This attribute can be changed.
- *hidden*
  Flag indicating whether the object is hidden. This is not the same as the invisible state. This attribute is intended to be used for rendering selections. This attribute can be changed.
- *incomplete*
  Boolean status flag indicating an aborted input operation during the read of the object from file, which returned the structure intact but without the complete set of associated data. An aborted input may be either be the result of an explicitly set input control flag, or by encountering property data which could not be decoded. This attribute is read-only.
- *invisible*
  Flag indicating whether the object is invisible. This is not the same as the *hidden* state. An invisible object is no longer accessible via its handle. This is usually the case for objects which are scheduled for deletion, but still have lingering pointer references. This attribute is read-only.

- *javaobject*
  If the toolkit was compiled with **JNI** support, this attribute reports the memory address of the **JNI** wrapper class instance, if it exists.
- *modcount*
  Object modification count. This attribute is read-only.
- *mutexcount*
  The number of recursive mutex locks held for this object. Only supported on Linux.
- *pyobject*
  If the toolkit was compiled with Python support, this attribute reports the memory address of the Python wrapper class instance, if it exists. This attribute is read-only.
- *pyrefcount*
  If the toolkit was compiled with Python support, this attribute reports the reference count of the Python wrapper class instance, if it exists. This attribute is read-only.
- *record*
  The current iterator record (starting with 1) of the ensemble. It is possible to set the value and thus skip or revisit ensemble molecules in the iterator.
- *refcount*
  If the **TCL** interpreter is using native Cactvs objects instead of string-based major object handles and integer-based minor object labels to identify toolkit objects, this returns the number of **TCL** object references active for this ensemble. This attribute is read-only.
- *scoped*
  A boolean object visibility control flag. If set, and global control flag `::cactvs(object_scope)` is also set, the object is visible only in the**TCL** interpreter which set the scope flag and thus claimed it. Object list commands executed in other interpreters omit this object, and attempts to decode its handle in other interpreters will fail. The most common use of this feature is the hiding of persistent chemistry objects in scripted property computation functions.
- *selected*
  Flag indicating whether the object is selected. This attribute can be changed.
- *tooltip*
  If the toolkit was compiled with factory support, this is the tooltip of the object icon on a workbench. This attribute can be changed.
- *uuid*
  An automatically generated **UUID** globally identifying the object. This attribute is read-only, different for every object, and not dependent on its contents.
- *x*
  f the toolkit was compiled with factory support, this is the x coordinate of the object icon on its workbench. This attribute can be changed.
- *y*
  If the toolkit was compiled with factory support, this is the y coordinate of the object icon on its workbench.This attribute can be changed.

ens getparam

```
ens getparam ehandle property ?key? ?default?
e.getparam(property=,?key=?,?default=?)
```

Retrieve a named computation parameter from valid property data. If the key is not present in the parameter list, an empty string is returned (`None` for PYTHON). If the default argument is supplied, that value is returned in case the key is not found.

If the key parameter is omitted, a complete set of the parameters used for computation of the property value is returned in dictionary format.

This command does not attempt to compute property data. If the specified property is not present, an error results.

Example:

```
ens getparam $ehandle E_GIF format
```

returns the actual format of the image, which could be *gif*, *png*, or various bitmap formats.

ens groups

```
ens groups ehandle ?filterset? ?filtermode?
e.groups(?filters=?,?mode=?)
```

Standard cross-referencing command to obtain the labels or references of the groups the ensemble contains. This is explained in more detail in the section about object cross-references.

Example:

```
ens groups $ehandle
```

ens hadd

```
ens hadd ehandle ?filterset? ?flags? ?changeset?
e.hadd(?filters=?,?flags=?,?changeset=?)
```

Add a standard set of hydrogens to the ensemble. If the *filterset* parameter is specified, only those atoms which pass the filter set are processed.

Additional operation flags may be activated by setting the *flags* parameter to a list of flag names, or a numerical value representing the bit-ored values of the selected flags. By default, the flag set is empty, corresponding to the use of an empty string or *none* as parameter value. These flags are currently supported:

- *keepflags*
  For expert use only. Do not discard min/max values and property scope flags for atom properties when hydrogen is added.
- *no2dcoords*
  Do not assign 2D coordinates to the added hydrogens, even if the rest of the atoms in the ensemble have valid 2D coordinates. In any case, 2D coordinates are never added when the ensemble does no already possess valid 2D coordinates.
- *no3dcoords*
  Do not assign 3D coordinates to the added hydrogens, even if the rest of the atoms in the ensemble have valid 3D coordinates. In any case, 3D coordinates are never added when the ensemble does no already possess valid 3D coordinates.
- *noanions*
  Do not add hydrogen to atoms with a negative formal charge.

- *noatoms*
  Do not add hydrogen to atoms without any bonds.
- *nocations*
  Do not add hydrogen to atoms with a positive formal charge.
- *noelements*
  Do not add hydrogen if the ensemble consists purely of isolated metal atoms, which probably represent the material in elementary form, or as an alloy.
- *noexcessvalences*
  Similar to *nohighvalences*, but hydrogen is not added to any atom which is not in its lowest standard bonded valence state.
- *nofixatomtext*
  Do not adjust property `A_TEXTLABEL` (if present) by removing references to implicit H from it on atoms where hydrogen is added. For example, by default “NHCOOEt” becomes “NCOOEt” after adding an instantiated hydrogen to the nitrogen atom. This reduces confusion on the hydrogen status when rendering all atoms.
- *nohighvalences*
  Do not add hydrogen to atoms which already exceed their lowest standard valence minus any formal charge. This option only applies to elements which have a defined lowest standard valence (this is configurable via the element table).
- *nomemory*
  Do not remember the added hydrogen atoms as automatically added. Normally, a flag is retained as part of the atom information which distinguishes atoms which were added by automatic processing, such as hydrogen addition, from those which were originally input.
- *nometals*
  Do not attempt to add hydrogen to atoms which are metals (as defined in the system element table).
- *nospecial*
  Do not perform hydrogen addition to atoms which participate in non-standard bonds (all bonds with `B_TYPE` not *normal*).
- *protonate*
  Add a single proton to the first suitable atom. The charge of the atom is increased, and only a single hydrogen is added regardless of the standard number of missing hydrogens,. This command *does* issue the standard property invalidation event for atom and bond changes. In the ensemble command variant, this option is rarely useful. It is supported for compatibility with the **`atom hadd`** command.
- *resetmemory*
  Reset the origin flag described above for all atoms in the ensemble. All current atoms act as if they were part of the original atom set.

Adding hydrogens with this command, except wit a set *protonate* flag, is less destructive to the property data set of the ensemble than adding them with individual **`atom create/bond create`** commands, because many properties are designed to be indifferent to explicit hydrogen status changes, but are invalidated if the structure is changed in other ways.

If the effects of the hydrogen addition step to the validity of the property data set should not be handled according to this standard procedure, it is possible to explicitly generate additional property invalidation events by specifying an event list as the optional last parameter, for example a list of *atom* and *bond* to trigger both the atom change and bond change events.

The command returns the number of hydrogens which were added.

Example:

```
set ehandle [ens create {[C].[C]}]
ens hadd $ehandle
```

adds a total of eight hydrogens to the two carbon atoms, transforming them into methane.

ens hdup

```
ens hdup ehandle ?datasethandle? ?position? ?filterset? ?ctonlyflag?
e.dup(?dataset=?,?position=?,?filters=?,?ctonly=?)
```

This command is a convenience variant of the **`ens dup`** command. It has the same parameters, but also adds a full standard hydrogen set (equivalent to executing an **`ens hadd $eh`** command) to the duplicate.

The command arguments are documented in the paragraph on **`ens dup`**.

ens hfragment

```
ens hfragment ehandle atomlist ?datasethandle? ?position?
e.hfragment(atomsequence=,?dataset=?,?position=?)
```

This command has the same arguments as **`ens fragment`**. The only difference is that after the duplication all open valences in the fragment are plugged with hydrogen, as if an **`ens hadd`** command had been executed immediately after the fragment creation command.

The command returns the handle or reference of the new ensemble object.

ens hierarchy

```
ens hierarchy ehandle ?filterlist? ?root?
e.hierarchy(?filters=?,?root=?)
```

Return the hierarchy handle or reference of the hierarchy the ensemble is part of. If the ensemble is not member of a hierarchy, or does not pass all of the optional filters, an empty string or **`None`** for **Python** is returned. By default, the hierarchy object which directly contains the ensemble is returned. If the *root* flag is set, the root hierarchy object is reported instead, which is the same only if the hierarchy has only a single level.

Example:

```
ens hierarchy $ehandle
```

ens hstrip

```
ens hstrip ehandle ?flags? ?changeset?
e.hstrip(?flags=?,?changeset=?)
```

This command removes hydrogens from the ensemble. By default, all hydrogen atoms in the ensemble are removed.

The *flags* parameter can be used to make the operation more selective. It may be a list of the following flags:

- *deprotonate*
  If this flag is set, a single proton is removed from the first suitable atom. This command variant *does* issue a standard atom and bond change property invalidation event, and it always ends processing after removing the first proton. Proton removal decreases the charge of the atom by one. In the ensemble command variant, this flag is rarely useful - it is supported for compatibility with the **`atom hstrip`** command
- *keepalphawedge*
  Keep hydrogen atoms which are bonded to an atom which is at the tip of a wedgebond. This flag excludes the case where the bond to the hydrogen atom is the wedge bond - use the *keepwedge* flag to cover this case.
- *keepisotopes*
  Keep hydrogen atoms which are isotope labels (including enriched/depleted ^{1}H).
- *keeporiginal*
  Hydrogen atoms which were not automatically added via a hydrogen addition command are retained. Note that these commands can be run in a mode which does not leave information about automatic addition - hydrogens added this way are not retained.
- *keepprotons*
  Keep any molecules which consist only of hydrogen atoms (such as protons, hydride anions, and molecular hydrogen).
- *keepspecia*l
  If this flag is set, hydrogens which are usually displayed, such as on aldehydes, wedge bonds, carbon triple bonds or hetero atoms are retained.
- *keepwedge*
  Keep hydrogens which are at the end of a wedge bond, indicating stereochemistry.
- *normalize*
  Normalize the wedge pattern for standard cases, removing excess wedges from hydrogens if the result structure is still stereochemically defined. Hydrogens which lose their wedge in this process are no longer protected by the *keepwedge* flag.
- *wedgetransfer*
  If a hydrogen atom is removed which is at the end of a wedge, the wedge information is saved by transferring the wedge (changing its up/down status if necessary) to an adjacent, surviving bond. This flag has no effects if the *keepspecial* or *keepwedge* flags are set. This flag is set by default.

If the *flags* parameter is an empty string, or *none*, it is ignored. The default flag value is *wedgetransfer* - but this default value is overridden if any flags are set!

If the *changeset* parameter is specified, the property change events listed in the parameter are triggered after the command.

Hydrogen stripping is not as disruptive to the ensemble data content as normal atom deletion, except when the *deprotonate* flag is set. The system assumes that this operation is done as part of some file output or visualization preparation. However, if any new data is computed after stripping, the computation functions see the stripped structure, and proceed to work on that reduced structure without knowledge that the structure may contain implicit hydrogens.

The command returns the number of stripped hydrogens.

Example:

```
ens hstrip $ehandle [list keeporiginal wedgetransfer]
```

ens hydrogenate

```
ens hydrogenate ehandle ?filterset? ?changeset?
e.hydrogenate(?filters=?,?changeset=?)
```

Reduce all bonds in the ensemble to single bonds, except those excluded by the filter set.

If a change set is supplied, its interpretation is the same as in **`ens hadd`**.

The command returns the number of added hydrogens.

Example:

```
ens hydrogenate $eh {!arobond !ccbond}
```

This reduces all non-aromatic bonds involving hetero atoms to single bonds.

ens image

```
ens image ehandle ?width? ?height? ?options?
```

This command generates a **Tk** image object displaying the ensemble as an icon. The command is only available in toolkit variants which are linked with the portable **Tk GUI** toolkit library and which are either statically linked with the **GD** image drawing library, or can load it dynamically. It is currently not support in the **Python** interface.

The default image size is 64x64 pixels, but this may be overridden by the *width* and *height* parameters. If only *width* is set, it is also used for the height. The command returns a **Tk** image handle. These images may for example be placed on **Tk** canvases as canvas objects, or used on buttons and other **GUI** objects.

Because of the small size of the images, atoms are not displayed as symbols, but small color-coded squares. This is a command for the implementation of graphical structure-handling applications with icons. For serious structure visualization, use the `E_GIF`, `E_EMF_IMAGE` or `E_EPS_IMAGE` properties.

Additional options may be added by an arbitrary sequence of option/value pairs. Color names can be those registered in the **X11** color database, or a numeric specification in the *#rrggbb* format. These options are currently supported:

- -background *color*
  Background color. The default is black.
- -border *npixels*
  Thickness of the image border. The default are 5 pixels.
- -bordercolor *color*
  Border color. The default is *blue*.

- -cmode *none/special/all*
  Display mode for carbon atoms. The default is *special*, meaning that only carbon atoms which usually are drawn with a C symbol are displayed as colored rectangle and not just a bond node. Highlighted atoms are always displayed.
- -highlightatom *label*
  Select an atom for highlighting. By default, no atom is highlighted.
- -highlightcolor *color*
  Set the highlighting color. The default is *chartreuse*.
- -hmode *none/special/all*
  Display mode for hydrogen atoms. The default is *special*, meaning that only hydrogen atoms which usually are drawn with an H symbol are displayed as colored rectangle. Other hydrogen atoms and the bonds leading to them are suppressed. Highlighted atoms are always displayed.
- -imagename *name*
  Explicitly set a name for the image. By default, a name of the form *image**n*** is automatically generated. It is possible to specify the name of an existing image, which will then be overwritten.
- -linecolor *color*
  Color of bond lines and wedges. The default is *white*.

Images are cached. If an image for the selected ensemble with the same display attributes exists, it is reused.

Example:

```
set img [ens image $ehandle 80 80 -border yellow -linecolor blue]
canvas create .canvaswin image 50 50 -image $img
```

ens index

```
ens index ehandle
e.index()
```

Get the position of the ensemble in the object list of its dataset. If the ensemble is not member of a dataset, -1 is returned.

ens isotopecheck

```
ens isotopecheck ehandle ?failedatomvariable? ?extended?
e.isotopecheck(variable=,extended=)
```

Test whether the isotope labels on the atoms of the ensemble, if they exist, are physically reasonable. The command returns the number of failed atoms. If a capture variable is specified, the atom labels or references of these atoms are stored therein. If no isotope labels are set in `A_ISOTOPE`, the command always reports zero problems.

By default, a smaller isotope table is used which contains only isotopes which are sufficiently long-lived to perform chemistry on. These include naturally occurring isotopes as well as isotopes used for experimental labeling, such as ^{3}H or ^{14}C. If the *extended* boolean flag is set, a larger table containing all known isotopes of the elements is used.

The *isocheck* command is an alias.

ens jget

```
ens jget ehandle propertylist ?filterset? ?parameterdict?
e.jget(property=,?filters=?,?parameters=?)
Ens.Jget(data,property=,?filters=?,?parameters=?)
```

This is a variant of **`ens get`** which returns the result data as a **JSON** formatted string instead of **TCL** interpreter objects. The command is usable only for property data, not attribute retrieval.

The **PYTHON** class method is a one-shot command. The transient ensemble created from the initialization items is automatically deleted when the command finishes.

ens jnew

```
ens jnew ehandle propertylist ?filterset? ?parameterdict?
e.jnew(property=,?filters=?,?parameters=?)
Ens.Jnew(data,property=,?filters=?,?parameters=?)
```

This is a variant of **`ens new`** which returns the result data as a **JSON** formatted string instead of **TCL** interpreter objects.

The **PYTHON** class method is a one-shot command. The transient ensemble created from the initialization items is automatically deleted when the command finishes.

ens jshow

```
ens jshow ehandle propertylist ?filterset? ?parameterdict?
e.jshow(property=,?filters=?,?parameters=?)
Ens.Jshow(data,property=,?filters=?,?parameters=?)
```

This is a variant of **`ens show`** which returns the result data as a **JSON** formatted string instead of **TCL** interpreter objects.

The **PYTHON** class method is a one-shot command. The transient ensemble created from the initialization items is automatically deleted when the command finishes.

ens ldup

```
ens ldup ?ehandlelist?...
Ens.Ldup(?eref/erefsequence?,...)
```

Duplicate all ensembles in the argument list(s) in default mode.

The return value is a single list (even if multiple source lists are used) of the duplicated ensemble handles or references. If an argument list element is an empty string (or **`None`** for **PYTHON**), it indicates a missing object, and the output list also receives an empty string element (for **TCL**) or **`None`** (for **PYTHON**) at its position, without raising an error.

ens lhdup

```
ens lhdup ?ehandlelist?...
Ens.Lhdup(?eref/erefsequence?,...)
```

Duplicate all ensembles in the argument list(s) in default mode, and add hydrogens.

The return value is a single list (even if multiple source lists are used) of the duplicated ensemble handles or references. If an argument list element is an empty string (or **`None`** for **PYTHON**), it indicates a missing object, and the output list also receives an empty string element (for **TCL**) or **`None`** (for **PYTHON**) at its position, without raising an error.

ens list

```
ens list ?filterlist?
Ens.List(?filters=?)
```

This command returns a list of the ensemble handles currently registered in the application. This list may optionally be filtered by a standard filter list. If the filter operates on ensemble minor objects such as atoms or bonds and not directly on the ensemble object, it is sufficient if a single minor object passes the filter.

Example:

```
ens list halogen
```

lists the handles of all ensembles in the application which contain one or more halogen atoms.

ens lock

```
ens lock ehandle propertylist/objclass/all ?compute?
e.lock(property=,?compute=?)
```

Lock property data of the ensemble, meaning that it is no longer managed by the standard data consistency manager. The data consistency manager deletes specific property data if anything is done to the ensemble which would invalidate the information. Blocking the consistency manager can be useful when building ensembles from components in a script. Property data remains locked until is it explicitly unlocked.

The property data to lock can be selected by providing a list of the following identifiers:

- Property names
  Valid property instances on the ensembles or ensemble minor objects are locked. If the boolean *compute* flag is set, an attempt is made to compute the property if it is not yet present. Otherwise, a request to lock non-existent data is silently ignored. It is not possible to lock individual property fields.
- *all*
  All valid ensemble and ensemble sub-object properties are locked. The compute flag is ignored.
- *ens,atom,bond,...*
  These is are object class identifiers. All property data which is controlled by the ensemble major object and attached to the specified object class is locked.

The lock can be released by an **`ens unlock`** command.

The return value is the original ensemble handle or reference.

Example:

```
set eh [ens create CCC]
ens lock $eh A_SYMBOL 1
ens purge $eh A_ELEMENT
```

```
atom set $eh 1 A_query(dsearch) 3
ens unlock $eh A_SYMBOL
```

In this example, an ensemble is created, and the atom symbol information is locked. Next, the element number property is deleted, and a query attribute is set. Finally, the lock is released. Had the element symbol information not been locked, the ensemble would have become unusable due to an overzealous data consistency manager. Setting query information in property `A_QUERY` *can* have an influence on the atom symbol. So the default action of invalidating `A_SYMBOL` when manipulating `A_QUERY` is correct. However, in case there is no element information `A_ELEMENT`, and no atom symbol information `A_SYMBOL`, the element information is completely lost, and the ensemble becomes unusable. So in this case, locking `A_SYMBOL` (or alternatively `A_ELEMENT`) is required to avoid unexpected side effects of structure editing.

ens loop

```
ens look ehandle objvariable ?maxmol? ?offset? body
e.loop(function=,?maxloop=?,?offset=?,?variable=?)
for m in e:
```

Loop over all molecules in the ensemble, by providing a temporary ensemble duplicate of each found molecule. The handle of the duplication is stored in the object variable and visible to the loop code.

The loop code cannot delete the duplicate ensemble. It is automatically deleted at the end of each cycle. Changes made to the duplicate molecule are not seen in the base ensemble. It is however possible to explicitly assign data computed on the duplicate ensemble to the base ensemble.

The optional parameters allow more control over which molecules are processed. By default the *maxmol* parameter is -1, meaning an unlimited number of fragments are processed, and the offset is zero, meaning that processing begins with the first molecule in the molecule list of the base ensemble.

For **Tcl** scripts, within the loop code, the standard **Tcl** commands *break* and *continue* work as expected.

The **Python** version of the loop method does intentionally have a different argument sequence for convenience. The function argument may either be a multi-line string (similar to the **Tcl** construct), or a function reference. Functions are called with the reference of the current loop object as single argument, and have their own context frame, so that the specification of a reference variable is not generally useful in that call style, though is is allowed. For string function blocks the code is executed in the local call frame, and the variable with the current object reference is visible locally. Script code blocks must be written with an initial indentation level of zero. Within the **Python** functions, the normal *break* and *continue* commands cannot be used to to scope limitations. Instead, the custom exceptions *BreakLoop* and *ContinueLoop* can be raised. These are automatically caught and processed in the loop body handler code.

In **Python**, there is also an object iterator so that simple loops over ensemble molecules can be written with a **`for`** statement. The ensemble object iterator is of the *self* style (i.e. there is one per ensemble, these are not independent objects), so nesting them is not possible on the same ensemble.

Python object loop constructs and their peculiarities are discussed in more detail in the general chapter on **Python** scripting.

The command returns the number of molecule fragments processed.

Example:

```
set midx 0
ens loop $ehandle ehdup {
   mol set $ehandle [mol mol $ehandle #$midx] M_MYPROP [ens get $ehdup E_MYPROP]]
   incr midx
}
```

The example loop assigns a custom property where the compute function is only defined for a single-fragment ensemble to the equivalent molecule property in a multi-fragment base ensemble.

ens mask

```
ens mask ehandle labellist/all property onvalue ?offvalue?
e.mask(objects=,property=,onvalue=,?offvalue=?)
e.mask(“all”,property=,onvalue=,?offvalue=?)
```

This command sets property values of a subset of minor objects of one class in the ensemble to a specific value, and optionally resets the values of the same property for all other minor objects of the ensemble which are not selected.

The first argument after the ensemble handle is either a list of object identifiers, or the magic value *all*. Object identifiers are usually the standard numerical labels, but any construct which identifies an atom, a bond, etc. can be used. The next argument identifies the property. The object identifiers in the previous argument must correspond to the object class of the property, i.e. atom label pairs can only be used it the property is a bond property, but simple numerical labels work for all classes. If data for that property is not present on the ensemble, it is instantiated with the default value. The final one or two arguments must be decodable data values for that property.

If the *all* object subset identifier is used, all values of the property in the ensemble are set to the *onvalue*. Any *offvalue* specification is ignored.

Otherwise, the explicit label list is processed. If an off value is given, all values of the property in the ensemble are first reset to that value. If no off value was specified is, no reset is performed and the current values remain valid. Then, all minor objects in the list are looked up from their labels or other identifiers, and their property value set to the *onvalue*.

Example:

```
ens mask $eh [ens atoms $eh carbon] A_COLOR green black
```

This command sets the `A_COLOR` property value for all carbon atoms in the ensemble to *green*, and all other atoms to *black*. This is shorter and more efficient then explicitly coding a loop of **`atom set`** statements.

The command returns the original ensemble handle or reference.

ens match

```
ens match ehandle ss_ehandle ?matchflags? ?ignoreflags? ?atommatchvar?
?bondmatchvar? ?molmatchvar?
e.match(substructure=,?matchflags=?,?ignoreflags=?,?atommatchvariable=?,
   ?bondmatchvariable=?,?molmatchvariable=?)
```

Check whether the ensemble matches a substructure. The substructure may be any structure ensemble, and even be in the same ensemble as the primary command ensemble.

The precise operation of the substructure match routine can be tuned by providing a standard set of match flags and feature ignore flags. The default match flag set has set bits for the *bondorder*, *atomtree* and *bondtree* comparison features, and an empty ignore set. If a flag set is specified as an empty string, the default set is used. In order to reset a flag set, an explicit *none* value must be used. The bit options of the match flag are explained in the documentation of the **`match ss`** command.

The command returns boolean 1 for a successful match, 0 otherwise. If an optional atom, bond, or molecule match variable is specified, it is set to a nested list of matching substructure/structure atom, bond or molecule labels (**Tcl**) or references (**Python**). If no match can be found, the variable is set to an empty list. In case only a bond or molecule match variable is needed, an empty string can be used to skip the unused match variable argument positions.

This is a very simple variant of substructure matching. The **`match ss`** command provides many more advanced match determination and match processing options.

ens max

```
ens max ehandle propertylist ?filterset?
e.max(property=,?filters=?)
```

Get the maximum values of the properties named in the *propertylist* parameter. The return value of the command is a list of the maximum property values. The objects whose property values are used for the determination of the maximum values may optionally be filtered by a standard filter set. If no objects pass the filter, the result is an empty string.

Example:

```
ens max $ehandle A_ELEMENT
```

computes the maximum element number in the ensemble.

ens merge

```
ens merge ehandle ?ehandle_list?...
e.merge(?eref/erefsequence?,...)
```

Merge a set of ensembles into one ensemble. All structure information is accumulated in the first (base) ensemble. Its handle remains unchanged. All other ensembles are destroyed. It is not possible to name an ensemble more than once in the argument lists, and ensembles cannot be merged with themselves.

The merged ensemble has a consistent property set for all minor objects. If the information content of the input ensembles varies, an attempt is made to compute the missing information for ensembles which do not have valid data for each individual property. If the computation fails, the property data is discarded for all merged objects. In addition, a *merge* property invalidation event is issued, which may lead to additional loss of property data. For surviving properties which have defined a merge update function, this function is then called and may perform additional data adjustments. For example, the `A_XY` 2D plot coordinate property merge function transforms the structure plot coordinates in the new ensemble to a uniform scale and arrange the coordinates for the atoms from the merged ensembles as a sequence of plots from left to right.

The return value of this command is a list of the new first atom labels or references for every merged ensemble, excluding the base ensemble. All minor object labels in the merged ensembles are re-assigned to avoid collisions. The new labels begin with the highest respective minor object label in use in the base ensemble plus one, and are thereafter assigned in sequence. In case an empty ensemble was merged, the list contains an empty string (**Tcl**) or **None** (**Python**) at its merge position.

The **ens add** command performs the same operation as the **ens merge** command, but merges duplicates of the input ensembles, thus preserving them.

Example:

```
ens merge [ens create CC] [list [ens create CCC.CCCC] [ens create C]]
```

Merge three ensembles into one. The new ensemble contains the molecules ethane, propane, butane and methane in that order.

ens metadata

```
ens metadata ehandle property ?field ?value??
e.metadata(property=,?field=?,?value=?)
```

Obtain property metadata information, or set it. The handling of property metadata is explained in more detail in its own introductory section. The related commands **ens setparam** and **ens getparam** can be used for convenient manipulation of specific keys in the computation parameter field. Metadata can only be read from or set on valid property data.

Valid field names are *bounds*, *comment*, *info*, *flags*, *parameters* and *unit*.

Examples:

```
array set gifparams [ens metadata $ehandle E_GIF parameters]
ens metadata $ehandle E_NAME comment "This is a CAS name in 1995 revision. The IUPAC
name, or any previous or later CAS revision name, look completely different."
```

The first line retrieves the computation parameters of the property `E_GIF` as keyword/value pairs. These are read into the array variable **gifparams**, and may subsequently be accessed as **$gifparams(format)**, **$gifparams(height)**, etc. The second example shows how to attach a comment to a property value.

ens min

```
ens min ehandle propertylist ?filterset?
e.min(property=,?filters=?)
```

Get the minimum values of the properties named in the *propertylist* parameter. The return value of the command is a list of the minimum property values. The objects whose property values are used for the determination of the minimum values may optionally be filtered by a standard filter set. If no objects pass the filter, the result is an empty string.

Example:

```
ens min $ehandle A_FORMAL_CHARGE xatom
```

gets the lowest value of the formal charge of a hetero atom in the ensemble.

ens mols

```
ens mols ehandle ?filterset? ?filtermode?
e.mols(?filters=?,?mode=?)
```

Standard cross-referencing command to obtain the label(s) of the molecule the ensemble contains as minor objects. This is explained in more detail in the section about object cross-references.

Examples:

```
ens mols $ehandle
ens mols $ehandle heterocycle
```

The first example simply returns a list of the labels of the molecules the ensemble contains as minor objects. Note that it is possible that there is more than one molecule in the ensemble - this is the reason why the command name is *mols*, not *mol*. The second example returns the molecule label(s) of all the molecules in the ensemble which contain one or more heterocycles. If there are no such molecules, an empty list is returned.

ens move

```
ens move ehandle ?datasethandle|remotehandle? ?position?
e.move(?target=?,?position=?)
```

Make the ensemble a member of a dataset, or remove it from a dataset. If the dataset handle or reference parameter is omitted, or is an empty string, or **None** for **PYTHON**, the object is removed from its current dataset. The dataset handle or reference may be the name of a remote dataset for moving object over a network connection.

If a target dataset handle or reference is specified, the ensemble is added to the dataset, if allowed by the acceptance bits of the dataset, and removed from any dataset it was member of before the execution of the command. By default the ensemble is added to the end of the dataset object list, but the final optional parameter allows the specification of an object list index. The first position is index zero. If the parameter value *end* is used, or the index is bigger than the current number of dataset objects minus one, the ensemble is appended as per the default. It is legal to use this command for moving ensembles within the same dataset.

Another special position value is *random* or *rnd*. This value moves to the object to a random position in the dataset. Using this mode with remote datasets is currently not supported.

The dataset handle cannot be a transient dataset.

The return value of the command is the dataset of the object prior to the move operation. It is either a dataset handle/reference, or an empty string (**TCL**) or **None** (**PYTHON**) if it was not member of a dataset.

This command interacts with the insert control mechanism of size-constrained datasets. More information is provided in the description of the *sizecontrol* dataset parameter.

Examples:

```
ens move $ehandle $dhandle 0
ens move $ehandle
```

In the first example, the ensemble is inserted as the first element in a dataset. The second line reverts this operation and removes the ensemble from the dataset.

This command can be used with a remote dataset descriptor. In that case, the ensemble is packed into a serialized object representation, transmitted over the network and restored as member of the remote dataset at the specified position. The local ensemble is deleted if the transfer succeeds.

Example:

```
ens move $ehandle blockbuster@server2:9998 end
```

This command moves the ensemble to the dataset which was set up as listener on port 9998 and pass phrase *blockbuster* on host *server2*. The local ensemble is deleted, and its copy is inserted at the end of the remote dataset.

ens mutex

```
ens mutex ehandle mode
e.mutex(mode)
```

Manipulate the object mutex.

During the execution of a script command, the mutex of the major object(s) associated with the command are automatically locked and unlocked, so that the operation of the command is thread-safe. This applies to toolkit builds that support multi-threading, either by allowing multiple parallel script interpreters in separate threads or by supporting helper threads for the acceleration of command execution or background information processing.

Going beyond this automatic per-statement protection, this command locks major objects for a period of time that exceeds a single command. A lock on the object can only be released from the same interpreter thread that set the lock. Any other threaded interpreters, or auxiliary threads, block until a mutex release command has been executed when accessing a locked command object. This command supports the following modes:

- *lock*
  Increase the recursive mutex lock count on the object. The command returns the current lock count after the command, excluding the transient single-command lock.
- *reset*
  Release all persistent locks on the object, if they exist.
- *test*
  Return the current persistent lock count on the object. This excludes the transient per-command lock.
- *unlock*
  Decrease the recursive lock count on the object. The command returns the current lock count after the command, excluding the transient single-command lock. Unlocking an object which has not been persistently locked results in an error.

There is no *trylock* command variant because the command already needs to be able to acquire a transient object mutex lock for its execution.

The command returns the current lock count.

ens need

```
ens need ehandle propertylist ?mode? ?parameterdict?
e.need(property=,?mode=?,?parameters=?)
```

Standard command for the computation of property data, without immediate retrieval of results. This command is explained in more detail in the section about retrieving property data.

The return value is the original ensemble handle or reference.

Examples:

```
ens need $ehandle A_XY recalc
ens need $ehandle E_EINECS_ID threaded
```

ens new

```
ens new ehandle propertylist ?filterset? ?parameterdict?
e.new(property=,?filters=?,?parameters=?)
Ens.New(data,property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **`ens get`** command. The difference between **`ens get`** and **`ens new`** is that the latter forces the re-computation of the property data, regardless whether it is present and valid, or not.

The **Python** class method is a one-shot command. The transient ensemble created from the initialization items is automatically deleted when the command finishes.

ens nget

```
ens nget ehandle propertylist ?filterset? ?parameterdict?
e.nget(property=,?filters=?,?parameters=?)
Ens.Nget(data,property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **`ens get`** command. The difference between **`ens get`** and **`ens nget`** is that the latter returns numeric data, even if symbolic names for the values are available.

The **Python** class method is a one-shot command. The transient ensemble created from the initialization items is automatically deleted when the command finishes.

ens nnew

```
ens nnew ehandle propertylist ?filterset? ?parameterdict?
e.nnew(property=,?filters=?,?parameters=?)
Ens.Nnew(data,property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading object data and attributes. It is explained in more detail in the section about retrieving property data.

For examples, see the **`ens get`** command. The difference between **`ens get`** and **`ens nnew`** is that the latter always returns numeric data, even if symbolic names for the values are available, and that property data re-computation is enforced.

The **Python** class method is a one-shot command. The transient ensemble created from the initialization items is automatically deleted when the command finishes.

ens nitrostyle

```
ens nitrostyle ehandle style
e.nitrostyle(style=)
```

Change the internal encoding of nitro groups and similar functional groups in the ensemble. Possible values for the style parameter are:

- *asis* No change
- *ionic* Change to encoding to a positive charge on the center atom, and a negative on one of the oxygens
- *xionic* As above, but also change the encoding of azides, etc.
- *neutral* Change the encoding to the neutral form with extended valence. *pentavalent* is an alias.
- *xneutral* As above, but also change the encoding of azides, etc.

The command returns the original ensemble handle or reference.

ens op2d

```
ens op2d ehandle mode ?atomfilter_bit/degrees?
e.op2d(mode=,?atomfilter=?)
```

Perform various operations on the standard 2D layout coordinates of the structure (property `A_XY`). Properties tightly connected to `A_XY` are also updated (most notably, `B_FLAGS` to keep wedges in sync with stereochemistry defined in other properties).

In mode *rotate*, the optional argument is the rotation angle in degrees. If it is not specified, the default are 30 degrees.

For alignment and flipping operations, the atoms which are used to determine the orientation can be filtered by specifying one or more value bits of property `A_FLAGS`. Only atoms where one or more of these bits are set in `A_FLAGS` are used for computing the alignment (in modes *xalign*, *yalign*, *xyalign* - all atoms are moved) or are flipped (modes *hflip*, *vflip* - unselected atoms are not moved). If no but filter values are specified, or *none* is used, all ensemble atoms and bonds are processed.

The following modes are supported:

- *rotate*
  Rotate the 2D structure coordinates counterclockwise.
- *hflip*
  Perform a horizontal flip around the X axis, while maintaining stereochemistry.
- *vflip*
  Perform a vertical flip around the Y axis, while maintaining stereochemistry.
- *xalign*
  The largest eigenvector of the unweighted XY coordinates of the selected atoms is aligned with the X axis.
- *xyalign*
  The largest eigenvector of the unweighted XY coordinates of the selected atoms is aligned with the XY diagonal.
- *yalign*
  The largest eigenvector of the unweighted XY coordinates of the selected atoms is aligned with the Y axis.

Additionally, the mode argument may an ensemble handle or reference. In that case, it is interpreted as a substructure, matched onto the ensemble, and if a match is found, the 2D coordinates of the ensemble atoms are adjusted by scaling and rotation for maximum overlap between the 2D coordinates of the substructure and the matched part of the ensemble. This mode retains the relative positions of the matched atoms - this is not a full redraw operation around a match template.

The command returns 0 (nothing done) or 1 (coordinates changed).

ens pack

```
ens pack ehandle ?maxsize? ?requestprops? ?suppressedprops? ?compressionlib?
e.pack(?maxsize=?,?requestprops=?,?suppressedprops=?,?compressionlib=?)
```

Pack the ensemble object into a base64-encoded compressed serialized object string. This string does not contain any non-printable characters and is a full dump of the internal state of the object, omitting only property data that was declared to be so easily re-computed that a dump is not worthwhile. Outside object relationship information, such as the dataset the ensemble might be a member of, or tables the ensemble is associated with, are not included.

The maximum size of the object string (default -1, meaning unlimited) can be configured by the optional *maxsize* parameter. The size is specified in bytes. If the pack string would be longer than the maximum size, an error results.

The two optional parameters lists allow to request a specific property set to be part of the package, even if it normally would not be included, and to explicitly omit properties from the dump. No property computation is performed, and suppressed properties are not purged from the source ensemble.

Ensembles can be restored from a packed object string by the **`ens unpack`** and **`ens create`** commands.

The ensemble object and its minor objects are unchanged after using this command.

The default compression library is *zlib*. Other useful variants include *lzo* and *gzip* (and there are other internal types), but these may not be available on all builds due to license issues, and you need to specify the compression library when a dataset is unpacked. It is generally recommended to stay with *zlib*.

The return value of this command is the packed string.

In **PYTHON**, ensembles support the standard *pickle*/*unpickle* protocol.

Example:

```
set dbstring [ens pack [ens create CC=O]]
```

ens pis

```
ens pis ehandle ?filterset? ?filtermode?
e.pis(?filters=?,?mode=?)
```

Standard cross-referencing command to obtain the labels or references of the π systems the ensemble contains. This is explained in more detail in the section about object cross-references.

Examples:

```
ens pis $ehandle
```

π systems are a rather exotic feature and not commonly used. These are essentially descriptions of bonding interactions which use p or d orbitals, such as in standard covalent multiple bonds. A simple double bond is described with one σ system and one π system in this representation.

ens prepare

```
ens prepare ehandle molfilehandle
e.prepare(molfileref)
```

Prepare the ensemble for output via the specified file handle, for example by pre-computing properties that are needed for output. This has only an effect if the I/O module for the format of the file handle provides an output object preparation function, which is currently only the case for the **BDB** database format. The output of prepared and unprepared ensembles sent to the same file handle is indistinguishable.

The purpose of this command is to allow the preparation of the ensembles for output in a separate thread. For unprepared ensembles, a significant part of the time to write the record may be spent in computing required data. During this time, the file handle is blocked. Prepared ensembles already contain all required data, and are thus faster to write to file. The total time required in single-thread scripts for a simple **`molfile write`** command vs. a **`ens prepare`** plus **`molfile write`** combo is not much different. However, these operations are largely independent, and on multi-threaded scripts the total time savings can be significant if the two commands are executed in different threads.

The command returns the molfile handle or reference.

ens properties

```
ens properties ehandle ?pattern? ?noempty?
e.properties(?pattern=?,?noempty=?)
```

Get a list of valid properties of the ensemble and its minor objects. Property subsets may be selected by a non-empty filter pattern, which the property names must match in order to be listed. If the ensemble is a member of a reaction, reaction properties are included in the list. The same mechanism is used for dataset properties.

If the *noempty* flag is set, only properties where at least one data element controlled by the ensemble (i.e. a value for an atom of the ensemble, etc.) is not the property default value are output. By default, the filter pattern is an empty string, and the *noempty* flag is not set.

This command may also be invoked as **`ens props`** or **`e.props()`**.

Example:

```
ens properties $ehandle X_*
ens props $ehandle
```

The first example returns a list of the currently valid reaction properties of the reaction the ensemble is a member of, or an empty list if it is not. The second example lists all properties, including those of the ensemble proper, its minor objects such as atoms and bonds, and possibly of the reaction the ensemble is a member of, if it is an reaction ensemble.

ens purge

```
ens purge ehandle propertylist/objectclass/specialname ?emptyonly?
e.purge(?properties=?,?emptyonly=?)
```

Delete property data from the ensemble. The properties may either be properties of a reaction the ensemble is a member of (prefix `X_`), properties of a dataset the ensemble is a member of (prefix `D_`), or properties of the ensemble proper and its minor objects, such as ensemble or atom properties. If a property marked for deletion is not present, it is silently ignored.

If an object class name, such as *ens* or *atom*, is used instead of a property name, all properties of that class set on the ensemble are deleted, if they are not locked, or filtered out by the optional empty-only flag.

Setting the optional boolean flag *emptyonly* allows restricts the deletion to those properties where all the values for a property associated with a major object (such as on all atoms in an ensemble for atom properties, or just the single ensemble property value for ensemble properties) are set to the default property value.

Besides normal property names, a few convenient special names for common property deletion tasks are defined and can be used as a replacement for the property list. These include:

- *atomquery*
  Delete atom query properties (`A_QUERY` and any other atom query property).
- *atomstereochemistry*
  Delete all atomic atom stereo descriptors, but keep those for bonds.
- *bondquery*
  Delete bond query properties (`B_QUERY` and any other bond query property).
- *bondstereochemistry*
  Delete all bond stereo descriptors, but keep those for atoms.
- *isotopes*
  Delete isotope information in `A_ISOTOPE` and other isotope properties which may be defined in future software versions.
- *query*
  Delete query information (`A_QUERY` and `B_QUERY`, and any other query property).
- *radicals*
  Delete atomic radical information in `A_RADICAL` and other radical-related properties which may be defined in future software versions.
- *stereochemistry*
  Delete all stereochemistry descriptors, including 2D wedges, but not 3D coordinates. The implicit property list includes `A_LABEL _STEREO`, `B_LABEL_STEREO`, `A_CIP_STEREO`, `B_CIP_STEREO`, `A_DL_STEREO`, `B_CISTRANS_STEREO`, `A_HASH_STEREO`, `B_HASH_STEREO`, `A_MAP_STEREO`, `B_MAP_STEREO`, `A_STEREOINFO`, `B_STEREOINFO`, `A_STEREO_GROUP`, `M_STEREO_COUNT`, `E_STEREO_COUNT` and `B_FLAGS` (only selected bits, the property remains valid if present).
- *wedges*
  Delete wedge bond flags in property `B_FLAGS`. If `B_FLAGS` is not present, the command is ignored and no computation attempt is made.

Examples:

```
ens purge $ehandle X_IDENT
ens purge $ehandle E_IDENT 1
ens purge $ehandle stereochemistry
```

The first example deletes the property data `X_IDENT` from the reaction the ensemble is a member of - provided it actually is a reaction ensemble. The second example deletes property `E_IDENT` from the ensemble if the property value is equal to the default value for `E_IDENT`. The last example removes all stereochemistry information from the ensemble.

The command returns the original ensemble handle or reference.

ens reaction

```
ens reaction ehandle ?filterlist?
e.reaction(?filters=?)
```

Return the handle or reference of the reaction the ensemble is a member of. Optionally, the reaction may be filtered by a simple filter list. If the ensemble is not part of a reaction, or does not pass the filter, an empty string is returned for **Tcl**, and **`None`** for **Python**.

Because an ensemble can only participate in a single reaction, the command is spelled **`ens reaction`** in singular.

Example:

```
ens reaction $ehandle
```

ens rebuild

```
ens rebuild ehandle ?minor_objectclass?
e.rebuild(?objectclass=?)
```

This command discards all minor objects and attached property data of a specific class associated with the ensemble. Afterwards, the minor object set is re-populated by the standard set-up function of the object class, if such a set-up function is defined.

If no minor object class is specified, bonds are regenerated - for example from 3D atomic coordinates. *Bonds*, molecules (*mols*), sigma and pi systems (*sigmas*, *pis*), rings and ring systems (*rings*, *ringsystems*) can all be rebuilt. However, by default no reconstruction function is defined for *groups* and surface patches (*surfaces*), although it is possible to set one via the object class manipulation command.

Generally, object sets should only be regenerated under exceptional circumstances, for example in order to undo a manual manipulation. Object sets are automatically generated when they are required - for example, bonds are automatically derived from atomic 3D coordinates if any property data associated with bonds is used in any context, and the ensemble so far did not contain bond information. An explicit request to generate connectivity is rarely needed.

Under normal circumstances, the use of minor object information such as bonds encoded explicitly in an input file is preferable to indirectly derived sets, such as regenerated connectivity. The connectivity algorithm of the toolkit is rather capable, but has its limitations, especially when hydrogen-depleted charged structures are encountered.

Files encoded in a few notorious structure file formats, such as **PDB**, may contain an incomplete bond set - without any indication that the bond set is incomplete. The **PDB** input routine tries to detect this, and automatically augments the bond set if obvious deficiencies are found. However, in case of minor omissions in the input data, a **PDB** structure may be one of the rare cases when an explicit request for a rebuild of the bond set can be helpful.

Besides the set of ensemble minor objects, the pseudo object class *aro* is also recognized. This keyword triggers a re-evaluation of aromatic systems and re-assign Kekulé bond orders, but not completely redo the bond set.

Example:

```
ens rebuild $ehandle bonds
```

This command discards the old bond set, and generate a new one. This only works if there is information which can be used for regeneration, such as atomic 3D coordinates. If no such information is present, the loss of bonds is irreversible and the ensemble useless for almost all applications short of a simulated plasma torch atomization.

The command returns the original ensemble handle or reference.

ens ref

```
Ens.Ref(identifier)
```

Python only method to get an ensemble reference from a string handle or another identifier. For ensembles, other recognized identifiers are ensemble references, or integers encoding the numeric part of the handle string.

ens rename

```
ens rename ehandle srcproperty dstproperty
e.rename(srcproperty=,dstproperty=)
```

This is a variant of the **`ens assign`** command. Please refer the command description in that paragraph.

ens replace

```
ens replace ehandle property/enshandle/emptystring ?preserved_propertylist/all?
e.replace(source=,?keep=?)
```

Substitute the ensemble with a structure decoded from data held in an ensemble property of that ensemble, or with the structure and associated data of another ensemble identified by its handle.

The original handle of the command ensemble is always preserved. The original structure data, with the exception of explicitly saved properties, is discarded. If the structure source argument is an ensemble handle, that ensemble is deleted.

For convenience, the replacement data argument may also be an empty string, which results in a no-op.

If the replacement argument is a property name, the exact type of operation depends on the data type of the property. The following data types are currently supported:

- *structure*
  Replace command ensemble directly with the property data ensemble.
- *string*
  Try to interpret the string as a structure line notation (as in **`ens create`**).

- *url*
  Try to download the file behind the Internet address and read it as a structure file.
- *blob*
  Try to read the contents as an in-memory structure file record.
- *diskfile, mapfile*
  Try to read it as a single-record structure file.

Any other property data type, `NULL` values of the property, non-ensemble properties, or malformed data result in an error and the original structure remains unchanged.

The structure source property data does not become not a property of the updated ensemble. In that ensemble, by default all other ensemble properties of the original are also purged, and all ensemble properties of the replacement structure are retained. However, by specifying a list of properties to be transferred, or using the special argument *all*, all or a subset of the ensemble property data of the original ensemble can be transferred to the replacement structure and thus saved. Under these circumstances, property data from the original ensemble has precedence and overwrites existing values of the same property on the replacement ensemble. However, all ensemble property data on the replacement ensemble which are not overwritten remain present in the updated ensemble. It is not possible to transfer atom, bond, or any other ensemble minor object property data to the replacement structure directly with this command.

The command returns the original, unchanged ensemble handle or reference.

Examples:

```
ens replace $eh E_CANONIC_TAUTOMER [list E_IDENT E_NAME]
```

This command replaces the current structure with its canonic tautomer. The values of properties `E_IDENT` and `E_NAME` from the original ensemble are kept in the updated form, all other ensemble property data of the original is discarded.

```
ens replace $eh $ehnew
```

Replace the structure with the one in **`$ehnew`**. The second ensemble is destroyed in the process.

ens replicate

```
ens replicate ehandle ?count?
e.replicate(?count=?)
```

This command duplicates all molecules in the ensemble and appends them to the atom, bond and other minor object lists of the ensemble.

The default replication count is one, but any other number of duplications may be chosen by an appropriate *count* parameter. If the count is less than one, the command is silently ignored.

The command returns the original ensemble handle or reference. As part of the integration step, *merge* property invalidation events are generated.

The **`ens dup`** command generates a new ensemble, while this command expands the current ensemble.

Example:

```
echo [ens get [ens replicate [ens create C.CC]] E_SMILES]
```

This prints *C.CC.C.CC* as result SMILES string, because both molecules in the original ensemble were duplicated and appended to the existing ensemble data.

ens rings

```
ens rings ehandle ?filterset? ?filtermode?
e.rings(?filters=?,?mode=?)
```

Standard cross-referencing command to obtain the labels or references of the rings the ensemble contains. This is explained in more detail in the section about object cross-references.

Examples:

```
ens rings $ehandle
ens rings $ehandle [list heterocycle aroring]
```

The first example returns the labels of all rings the ensemble contains. If the ensemble does not contain any rings, an empty list is returned. Only labels of rings in the SSSR or ESSSR set are returned, even if the currently configured ring set is larger. The second example filters the rings - only heteroaromatic rings are reported.

ens ringsystems

```
ens ringsystems ehandle ?filterset? ?filtermode?
e.ringsystems(?filters=?,?mode=?)
```

Standard cross-referencing command to obtain the labels or references of the ring systems the ensemble contains. This is explained in more detail in the section about object cross-references.

Examples:

```
ens ringsystems $ehandle
ens ringsystems $ehandle [list heterocycle aroring]
```

The first example returns the labels of all ring systems the ensemble contains. If the ensemble does not contain any ring systems, an empty list is returned. The second example filters the ring systems - a ring system label is included in the output list only if that ring system contains one or more hetero aromats.

ens rotate

```
ens rotate ehandle angle axis ?center? ?property?
e.rotate(angle=,axis=,?center=?,?coordinateproperty=?)
```

Rotate the ensemble in 3D space by manipulating property `A_XYZ`, or a custom atom float vector coordinate property.

The *angle* argument is a floating-point number in degrees. The *axis* argument is a 3D vector in standard notation, i.e. usually a list/tuple of three floating point numbers for the *x*, *y* and *z* components. If the last optional argument is omitted, the center of rotation is the 3D unweighted coordinate average of all ensemble atoms with valid 3D coordinates, which is computed as property `E_CENTER`. If the *center* argument is specified, it is expected to be a 3D point which is used as center of rotation instead.

This operation triggers a *3dglop* property invalidation event.

The command returns the original ensemble handle or reference.

Example:

```
ens rotate $eh 60 {0 0 1}
```

Rotate the ensemble 60 degrees counterclockwise around the *z* axis.

ens scan

```
ens scan ehandle expression/queryhandle ?mode? ?parameterdict?
e.scan(query=,?resultmode=?,?parameters=?)
```

Perform a query on the ensemble object. The syntax of the query expression and the optional selection list is the same as that of the **`dataset scan`** command with a transient dataset consisting of the current ensemble only. For more details, please refer to the paragraphs on **`dataset scan`** and **`molfile scan`**.

The return value depends on the mode. The default query mode, this is different from the default in **`molfile scan`**, is *exists*.

ens set

```
ens set ehandle ?property value?...
e.set(property,value,...)
e.set({property:value,...})
e.property = value
e[property] = value
```

Standard data manipulation command for setting property data. It is explained in more detail in the section about setting property data.

Example:

```
ens set $ehandle E_NAME "Pharmacon X-25"
```

ens setparam

```
ens setparam ehandle property ?key value?...
ens setparam ehandle property dictionary
e.setparam(property,?key,value?...)
e.setparam(property,dict)
```

Set or update a property computation parameter in the metadata parameter list of a valid property. This command is described in the section about retrieving property data. The current settings of the computation parameters in the property definition are not changed.

The return value is the updated property computation parameter dictionary.

Example:

```
ens setparam $ehandle E_GIF comment "Top Secret Lead Structure"
```

ens setup

```
ens setup ehandle ?minorobjclass?
e.setup(?objectclass=?)
```

Query the status of the minor object lists in the ensemble, or initialize one of these to an empty list.

If no class is specified, a dictionary with all currently registered minor object classes of the ensemble is returned. The object class names are the key, the value is a boolean flag for the status.

If an object class argument is supplied, the object class is instantiated on the ensemble, if necessary by auto-loading an object class handler module. Unknown object class names result in an error. If the minor object class is already instantiated, it is not changed. Otherwise, an empty minor object set is added. This is even the case if the minor object class handler provides a default object setup function (see **`ens rebuild`** command). Instantiating an object class with this command always creates an empty collection of the minor objects associated with the ensemble.

Minor object lists are usually implicitly instantiated, as in

```
ens get $eh M_LABEL
```

which automatically sets up the molecule/fragment object set if it is not yet present, and populates it with objects identifying disconnected fragments in the ensemble, or

```
group create $eh [list $a1 $a2 $a3]
```

which adds a group to the ensemble, again automatically initializing the group object set if it was not initialized.

The **`ens setup`** command is intended for special circumstances and not commonly used.

ens show

```
ens show ehandle propertylist ?filterset? ?parameterdict?
e.show(property=,?filters=?,?parameters=?)
Ens.Show(data,property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **`ens get`** command. The difference between **`ens get`** and **`ens show`** is that the latter does not attempt computation of property data, but raises an error if the data is not present and valid. For data already present, **`ens get`** and **`ens show`** are equivalent.

The **Python** class method is a one-shot command. The transient dataset created from the initialization items is automatically deleted when the command finishes.

ens sigmas

```
ens sigmas ehandle ?filterset? ?filtermode?
e.sigmas(?filters=?,?mode=?)
```

Standard cross-referencing command to obtain the labels or references of the σ systems the ensemble contains. This is explained in more detail in the section about object cross-references.

Examples:

```
ens sigmas $ehandle
```

σ systems are a rather exotic feature and not commonly used. These are essentially descriptions of bonding interactions which use s orbitals, such as normal, covalent single bonds, or the central bond in multiple bonds. A simple double bond is described with one σ system and one π system in this representation.

ens sort

```
ens sort ehandle ?sort_property? ?relabel? ?duplicate? ?datasethandle? ?position?
e.sort(?property=?,?relabel=?,?duplicate=?,?target=?,?position=?)
```

Sort the atoms in an ensemble according to a property value. The default property is `A_LABEL`, the standard atom label. The first optional argument can be used to sort on a different property, or a property field. However, the property must be either an atom property, or a molecule property. If the *relabel* flag is set, the ensemble atoms and molecules are renumbered after the sort in ascending order, starting with one. By default, atoms and molecules retain their original labels even if they change positions. If the *duplicate* flag is set, the sort operation works on a duplicate of the original ensemble. If the flag is unset, or the argument omitted, the operation modifies the original ensemble object.

The final two optional arguments allow the direct transfer of the modified ensemble or duplicate into a dataset, similar to an **`ens move`** command. The ensemble may be inserted into a specific position of a target dataset. If the special value *end* is used, or the zero-based position index is beyond the current end of the target dataset, the ensemble is simply appended. By default the ensemble is not moved, and if it is moved without an explicit position, it is appended.

The sequence of the atoms in the ensemble is rearranged so that the atoms are in ascending order of the values of the sort property or property field. Indirectly, molecules are also rearranged to correspond to the sequence of the first atoms in every molecule. This operation triggers a *shuffle* property invalidation event. If the renumbering option is selected, the atom and molecule sets are re-labeled with their standard label properties (i.e. `A_LABEL` for atoms, `M_LABEL` for molecules) in ascending order, starting with one. Other minor object collections remain in their original sequence and retain their current labels. Certain important properties which, if present, are dependent on atom label values, notably `A_LABEL_STEREO`, `B_LABEL_STEREO` and `B_FLAGS`, are specifically adjusted to the new labeling scheme instead of being invalidated.

The command returns an ensemble handle or reference. If the operation was operating on a duplicate, it is the handle or reference of the new ensemble, otherwise that of the original ensemble.

ens split

```
ens split ehandle ?minsize? ?splitproperty?
e.split(?minsize=?,?splitproperty=?)
```

Split the molecules of the ensemble into individual ensembles. The return value is a list of the handles or references of the new ensembles. If the original structure contains only a single fragment, the result is the same as a simple **`ens dup`** command. The split structures do not become a member of a reaction or dataset, even if the original structure is.

The optional *minsize* parameter is a minimum value for the number of heavy atoms (property `M_HEAVY_ATOM_COUNT`) in the molecules. If this is not an empty string, molecules which have less heavy atoms than the minimum are not duplicated. If all molecules in the input ensemble are smaller than the required size, an empty list is returned.

The optional *splitproperty* argument can be used to split the ensemble on values of a molecule property, which needs to be either already set or computable, instead of simply separating fragments on connectivity. All molecules in the input ensemble which have a common value of this property are put into a joint result ensemble, and each distinct split property value starts a new result ensemble. Molecules with a common property value do not need to be present in the input ensemble in a consecutive sequence, nor are there any special requirements for the data type or value range of the split property, as long as the data type has a comparison function. If the values of the split property are distinct over all molecules in the input ensemble, the outcome of command is indistinguishable from running it without any split property.

Example:

```
lassign [ens split [ens create "CC.CC"]] eh1 eh2
```

This example creates an ensemble with two ethane molecules, splits it, and assigns the two new ensemble handles to variables *eh1* and *eh2*.

```
set elist [ens split $eh {} M_REACTION_LABEL]
```

Split ensemble along the original reagent or product data blocks found in an **RXN** or **RDF** file.

ens sqldget

```
ens sqldget ehandle propertylist ?filterset? ?parameterdict?
e.sqldget(property=,?filters=?,?parameters=?)
Ens.Sqldget(data,property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **`ens get`** command. The differences between **`ens get`** and **`ens sqldget`** are that the latter does not attempt computation of property data, but initializes the property value to the default and returns that default, if the data is not present and valid; and that the **SQL** command variant formats the data as **SQL** values rather than for **TCL** or **PYTHON** script processing.

The **PYTHON** class method is a one-shot command. The transient dataset created from the initialization items is automatically deleted when the command finishes.

ens sqlget

```
ens sqlget ehandle propertylist ?filterset? ?parameterdict?
e.sqlget(property=,?filters=?,?parameters=?)
Ens.Sqlget(data,property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **`ens get`** command. The difference between **`ens get`** and **`ens sqlget`** is that the **SQL** command variant formats the data as **SQL** values rather than for **TCL** or **PYTHON** script processing

The **PYTHON** class method is a one-shot command. The transient dataset created from the initialization items is automatically deleted when the command finishes..

ens sqlnew

```
ens sqlnew ehandle propertylist ?filterset? ?parameterdict?
e.sqlnew(property=,?filters=?,?parameters=?)
Ens.Sqlnew(data,property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **`ens get`** command. The differences between **`ens get`** and **`ens sqlnew`** are that the latter forces re-computation of the property data, and that the **SQL** command variant formats the data as **SQL** values rather than for **TCL** or **PYTHON** script processing.

The **Python** class method is a one-shot command. The transient dataset created from the initialization items is automatically deleted when the command finishes.

ens sqlshow

```
ens sqlshow ehandle propertylist ?filterset? ?parameterdict?
e.sqlshow(property=,?filters=?,?parameters=?)
Ens.Sqlshow(data,property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **`ens get`** command. The differences between **`ens get`** and **`ens sqlshow`** are that the latter does not attempt computation of property data, but raises an error if the data is not present and valid, and that the **SQL** command variant formats the data as **SQL** values rather than for **Tcl** or **Python** script processing.

The **Python** class method is a one-shot command. The transient dataset created from the initialization items is automatically deleted when the command finishes.

ens subcommands

```
ens subcommands
dir(Ens)
```

Lists all subcommands of the **`ens`** command. Note that this command does not require an ensemble handle.

ens surfaces

```
ens surfaces ehandle ?filterset? ?filtermode?
e.surfaces(?filters=?,?mode=?)
```

Standard cross-referencing command to obtain the labels or references of surface patches the ensemble contains. This is explained in more detail in the section about object cross-references.

Example:

```
ens surfaces $ehandle carbon
```

This example lists all surface patches which are associated with carbon atoms. Surface patches associated with other atoms, or with no atoms, are not listed.

ens swapin

```
ens swapin ehandle
e.swapin()
```

Swap an ensemble from the disk store fully back into memory, and disable further automatic loading and shelving. If the ensemble was not swapped out, the command does nothing.

The command returns the original ensemble handle or reference.

ens swapout

```
ens swapout ehandle
e.swapout()
```

Remove most of the ensemble data from memory and store it in a temporary disk store. The ensemble handle remains valid. As soon as it is used in a command again after this command has been executed, the swapped ensemble data is automatically reloaded from file, and then stored again when the object lock is released. To disable the automatic swapping of an ensemble, use the **`ens swapin`** command.

This command is intended to be used in cases where a large number of ensembles must be kept in memory. Its routine use is not encouraged - it is only useful in case the programmer knows about access patterns. In other cases, the standard virtual memory mechanism of the operating system might yield better performance results.

The ensembles are stored as binary blobs in a key/value store in a process-specific swap directory *cactvs%d,* (*%d* is replaced by the process ID) which is created automatically in the standard temporary directory. When an ensemble is deleted, its swap record is also removed, if one was created during the lifetime of the ensemble. When a **CACTVS** application program exits, the swap store as well as the swap directory are automatically deleted, even without explicit deletion of the last set of ensembles in memory. In case of program crashes, the swap directory and its contents may however survive. If ensemble swapping is used with unstable applications, the temporary directory should be checked from time to time.

The command returns the original ensemble handle or reference.

Example:

```
ens swapout $ehandle
```

ens tables

```
ens tables ehandle ?filterlist?
e.tables(?filters=?)
```

Return a list of the handles of all table objects the ensemble is associated with. Optionally, the table set may be filtered by a simple filter list. If the ensemble is not related to any table, or none of these tables passes the filter list, an empty string is returned.

This command is only available if the toolkit was compiled with table support.

```
Example:
ens tables $ehandle
```

ens taint

```
ens taint ehandle propertylist/changeset ?purge?
e.taint(property=,?purge=?)
```

Issue a property data tainting event which acts on the ensemble data.

If the ensemble is a member of a dataset, the dataset and its objects are *not* tainted.

The event list may contain any number of the following items:

- A property name.
  In that case, all properties which depend on the specified one are invalidated. If the optional *purge* parameter flag is also set, the specified property itself is also deleted. By default the self-deletion flag is not set.

- An object class
  All properties which a sensitive to changes in the object class collection associated with the target ensemble are deleted. Example:
  ```
  ens taint $eh atom
  ```
  This deletes all properties which are sensitive to changes in the atom make-up of the ensemble.
- *2dop*
  All properties which are dependent on 2D layout coordinates are invalidated.
- *3drelative*
  All properties which are dependent on relative inter-atomic 3D atomic coordinate changes are invalidated.
- *3dabsolute*
  All properties which are dependent on absolute 3D atomic coordinate changes are invalidated.
- *dup*
  All properties which do not survive duplication of the underlying object are invalidated.
- *hadd*
  All properties which are sensitive to hydrogen addition or deletion via dedicated hydrogen processing commands, which do not trigger the default atom and bond change events associated with atom addition or deletion and bond changes, are purged.
- *merge*
  All properties which are invalidated by merging ensembles are invalidated.
- *shuffle*
  All properties which are dependent on the order of minor objects in the ensemble are purged.
- *stereo*
  All properties which are invalidated by stereo changes are dropped.

The command returns the original ensemble handle or reference.

ens torsions

```
ens torsions ehandle ?filterset? ?filtermode?
e.torsions(?filters=?,?mode=?)
```

Standard cross-referencing command to obtain the labels or references of the torsion objects the ensemble contains as minor objects. This is explained in more detail in the section about object cross-references.

ens transfer

```
ens transfer ehandle propertylist ?targethandle? ?targetpropertylist?
e.transfer(properties=,?target=?,?targetproperties=?)
```

Copy property data from one ensemble to another ensemble or other major object, without going through an intermediate scripting language object representation, or dissociate property data from the ensemble. If a property in the argument property list is not already valid on the source ensemble, an attempt is made to compute it.

If a target object is specified, and a property is not an ensemble but an ensemble minor object property, the number of property-associated minor objects is usually expected to be the same in both ensembles, and expected to have the same label set, tough it is not required that they are in the same sequence. Property data is assigned to the target ensemble minor objects with the minor object label as reference key. In case of a label set or object count mismatch between the two ensembles, no error is raised. Excess source data items are discarded, and excess target minor objects, or those with unmatched labels, retain their original value if the property was present on the target, or are set to the default value if the property was freshly instantiated. In this command mode, the return value is the handle of the target ensemble. Source and target ensembles cannot be the same object.

If a target property list is given, the data from the source is stored as content of a different property on the target. For this, the data types of the properties must be compatible, and the object class of the target property that of the target object. No attempt is made to convert data of mismatched types. In case of multiple properties, the source property list and the target property list are stepped through in parallel. If there is no target property list, or it is shorter than the source list, unmatched entries are stored as original property values, and this implies that the object class of the source and target objects are the same.

If no target object is specified, or it is spelled as an empty string or **PYTHON** **`None`**, the visible effect of the command is the same as a simple **`ens get`**, i.e. the result is the property data value or value list. The property data is then deleted from the source object. In case the data type of the deleted property was a major object (i.e. an ensemble, reaction, table, dataset or network), it is only unlinked from the source object, but not destroyed. This means that the object handles returned by the command can henceforth the used as independent objects. They can be deleted by a normal object deletion command, and are no longer managed by the source object..

Properties which are ensemble minor object properties can only be transferred to another ensemble. Ensemble properties can be moved to other major objects.

Example:

```
ens transfer $eh E_EMF_IMAGE $eh2
```

This copies property `E_EMF_IMAGE` from the first ensemble to the second. The property data remains valid on the source ensemble.

```
set ehc [ens transfer $eh E_CANONIC_TAUTOMER]
```

Get the handle of the canonic tautomer of the source ensemble, and dissociate it from the source ensemble.

ens transform

```
ens transform ehandle SMIRKSlist ?direction? ?reactionmode? ?selectionmode?
   ?flags? ?overlapmode? ?{?exclusionmode? excludesslist}? ?maxstructures?
   ?timeout? ?maxtransforms? ?niterations? ?statusvariable?
```

```
e.transform(transforms=,?direction=?,?reactionmode=?,?selectionmode=?,?flags=?,
   ?overlapmode=?,?excludess=?,?maxstructures=?,?timeout=?,?maxtransforms=?,
   ?iterations=?,?statusvariable=?)
```

This command applies one or more **SMIRKS** transforms to an ensemble and returns a list of ensemble handles or references of transformation products. The transformation products are filtered for duplicates. The original start structure is never returned - if a transform set does not match at all, an empty list is returned.

The required parameter after the ensemble handle is a list of **SMIRKS** lines, where each **SMIRKS** line is itself a list. A **SMIRKS** line is in the simplest case a simple **SMIRKS** transform without any extra data, but it may be padded by additional parameters which apply only to the application of that transform. If these optional parameters local to the current transform are not specified, their global counterpart on the command line is used instead. The syntax of an individual **SMIRKS** line is

```
SMIRKStransform ?step? ?direction? ?flags? ?overlapmode?
```

The **SMIRKS** transform part is the only required list element. It may be provided either as a string in standard Daylight notation, or as a handle of a reaction, which should have been decoded in **SMIRKS** mode (see **`reaction create`** command). Care should be taken to pass **SMIRKS** strings as proper elements of a list, even if only a single string is used, because they may contain whitespace and naming information after the actual transform code. Example:

```
ens transform $ehandle [list [list {[C:1][C:2]>>[C:1]=[C:2] Dehydrogenation} 1]]
```

The string *Dehydrogenation* is part of the transform specification string and not the transform step. The name string is attached to the (intermediate, in this case) transform reaction object as property `X_NAME` and can be used to track the reaction history of transform result structures.

The optional **`step`** element in a transform line (a positive integer or 0) identifies the reaction step of the transform. Transform sets of different step numbers are isolated from each other and do not interact. Transforms are executed in ascending step number. Transforms with different step numbers need not to be sorted, and the step numbers neither need to begin with one, nor form an uninterrupted sequence. A step number of 0 disables the transform. The default step number is one. All transforms of the same step number are essentially executed in parallel and may interact with each other.

The third and again optional element of transform lines is the direction identifier. It may be either *forward*, *backward*, or *bidirectional*. In *forward* mode, only the left part of a transform is used for matching, and the matched structure part is modified according to the description on the right side. *backward* works the other way around, and in *bidirectional* mode, both sides of the transform scheme are independently matched, and, if the match is successful, transformed to the other side. If this parameter is not specified, or specified as an empty string, the global *direction* parameter from the command line is substituted.

The fourth and once more optional element of a transform line is a list of flag words. Every word sets an additional flag. Currently, the following flag words are recognized:

- *absolutestereo*
  If this flag is set, the stereochemistry of the right side of a transform is transferred unchanged to the transform result ensemble, without attempting to interpret the operation as a reaction with stereochemistry inversion or retention by examination of the pattern on the left side. If the left side does not contain stereochemistry, the behavior induced by this flag is already the default and it has no effect. It also has no effect if the right side of the transform does not specify stereochemistry.
- *allrequired*
  If this flag is set, only result structures which were generated by the combined application of all transforms marked with this flag are accepted as final results. If any of the transforms marked with this flag did not contribute to a result structure, it is discarded. By default, the result set is not filtered by its origination from any specific transform.

- *anyrequired*
  If this flag is set, only result structures which were generated by the application of at least one of all transforms marked with this flag are accepted as final results. By default, the result set is not filtered by its origination from any specific transform.
- *appendpathname*
  The same as *setpathname*, except that the content of an existing property `E_NAME` on the input ensembles is not overwritten. Transform path information is always appended. If the input structure does not have initial name information, the operation of the two flags is indistinguishable.
- *changeelements*
  If set, the element number and atom type of matched atoms is changed to that of the matching right side template. By default, atom type and element number of atoms which are not newly added are preserved in the transformed ensembles. This is usually desirable for the use of element lists and other generic expressions as part of transform patterns. If this flag is set, the atom is changed to the exact template definition - including changes to *any* atoms, element lists, or complete atomic recursive **SMARTS** expressions.
- *chargeneutral*
  If set, the sum of all changes in the formal atom charges in the set modified by the application of the transform, excluding any atoms which are deleted or added, must be zero. This is helpful for example for charge redistribution transforms. For example, a transform like

  ```
  [*;+1:1]=[*:2][N:3]>>[*:1][*:2]=[N;+1:3]
  ```

  only works on structures where the nitrogen atom is neutral, because otherwise the total charge of the match three atom block would change. It would be possible to achieve the same effect with explicit indication of allowed charges on all involved atoms, but this flag can be convenient.
- *chargeradicals*
  If this flag is set, radicals which are generated as result of a transform are charged using chemistry common sense. A cleaner and preferable method is to explicitly encode charge in the transform.
- *checkaro*
  If set, aromaticity checking takes place. Atoms specified as aromatic in the transform pattern only match aromatic atoms in the target ensemble, and all other atoms only match non-aromatic atoms. By default, the aromaticity status of atoms is ignored in evaluating the pattern match.
- *checkcharges*
  If set, formal charges on the match side of the transform must exactly match the charges on the matched structure atoms. By default, charges are not used for determining a match. This flag should be set if the transform pattern should only match specific charges.
- *checkkekule*
  If this flag is set, bond orders of aromatic systems in the substrate molecules must be matched exactly as specified in the transform.
- *checkstereo*
  If set, the stereochemistry on the match side of the transform must match the stereochemistry on the matched structure atoms. By default, stereochemistry is not used for matching.

- *checkwedges*
  If this flag is set, bonds in the transform ensemble must match the wedge style specified in the left side of the transform template. This is useful only under very specific circumstances, since the style and placement of wedges does not uniquely identify a stereo isomer. Checking stereochemistry is therefore usually performed via the *checkstereo* flag, which relies on the comparison of stereo descriptors instead of wedges.
- *distinctpatternmatch*
  If this flag is set, the match mode of the substructure side of the transform is changed. The default match mode is *all*, meaning that all possible orientations of the substructure are generated, except in case of a transform application mode *first*, where the substructure match mode is also *first*. If this flag is set, the match mode is changed to *distinct*. In this mode, only pattern matches which differ in the set of structure atoms matched are generated, removing alternative mappings of the substructure on the same set of structure atoms. This mode is faster and can reduce the number of computation steps significantly, but the applicability of this match mode for the generation of the full set of desired transform results must be determined by the programmer with an eye for possible asymmetry of the matched structures outside the atom set of the transform substructure.
- *dropradicals*
  If this flag is set, transform result structures are discarded if they are radicals. In case the *chargeradicals* flag is also set, the radical check is performed after the attempt to charge standard radical centers and may thus be used as a second line of defense against unreasonable structures.
- *filtercharges*
  If set, use the localization of formal charges on atoms as a criterion to distinguish transformation results. By default, the standard hashcoding process is used which does not care about the placement of formal charges as long as these forms are interconvertible. For example, with the standard duplicate filtering process, pentavalent and ionic forms of nitro groups are considered equivalent. However, this will also prevent transforms which convert one form of a nitro group into another from working, since the transform result is discarded as being equivalent to the input structure. In order for this kind of transform to function as expected, the *filtercharges* flag must be set, which configures the duplicate filter to distinguish between the two forms. In that case, the *preservecharges* flag (see below) must not be set in order to allow the transformation to change the charge, but the *checkcharges* flag (see below) should be set in order to restrict the match of the transform to a specific ionic or pentavalent form.
- *filterisotopes*
  This flag instructs the duplicate detection mechanism to use hash codes which use isotope labeling information for duplicate removal. This flag is not exclusive to the *filterstereo* flag - both attributes can be combined to select a suitable hash code.
- *filterkekule*
  This flag instructs the duplicate detection mechanism to use compute hash codes which are dependent on the exact bond order - including that of Kekulé structures of aromatic systems - for duplicate removal.

- *filterradicals*
  If set, use the localization of free electrons on atoms as a criterion to distinguish transformation results. By default, the standard hashcoding process is used which does not care about the placement of electrons as long as these are interconvertible. However, this also prevents transforms which convert from one electron localization scheme to another without accompanying atom or bond changes from working, since the results are discarded as being equivalent to the input structure.
- *filterstereo*
  This flag instructs the duplicate detection mechanism to use stereo-specific hash codes for duplicate removal. This flag is not exclusive to the *filterisotopes* flag - both attributes are used to select a suitable hash code.
- *keepiterationintermediates*
  If this flag is set, and multiple iterations are run, the results from intermediate iteration steps are part of the returned set. By default, only the results of the last iteration are returned.
- *kekulize*
  If this flag is set, a new Kekulé form of aromatic systems in the transformed structure is constructed. This is useful when the matching pattern did not check for explicit single and double bonds, so that after applying the transformation the Kekule pattern may be wrong, for example after swapping an electron-pair donating N witch a normal pi-bonded C atom. Still, it is generally recommended to use explicit bond order manipulation since that method is more robust.
- *linkreaction*
  Ensembles which are created via a transform for which this flag is set are linked to an automatically created reaction object in which the transform result ensemble is the reaction product, and a duplicate of the input ensemble the reagent. In addition, the `X_NAME` and `E_REACTION_ROLE` properties are set. The return value of the **`ens transform`** command is still a list of the handles of the transform result ensembles. The additional reagent ensemble handles are not included, and neither are the handles of the reactions. In order to access the reaction information, a lookup command such as **`ens reaction`** with a result ensemble as argument can be used.
- *lockimplicitbonds*
  Bonds in the transform structure which are represented by bonds with an implicit bond order on the right side of the pattern (in forward direction, left side for reverse transforms) do not get their bond order adjusted, with the rationale that these pattern bond orders are not well defined anyway.
- *nitrostandardizer*
  If this flag is set, the input structure duplicate(s) entered as start compound in the transform processing queue is standardized to possess the neutral, pentavalent form of nitro groups and similar groups. This option does not change the input structure(s) of this command but only the first structure duplicate entered into the processing queue.
- *nochargepaircollapse*
  Disable the feature that bonds which connect atoms of opposite +1 and -1 formal charges are also matched by the equivalent bond with a bond order increased by one and neutral atoms.

- *nohadd*
  Part of the normal transformation procedure is a final hydrogen addition step before duplicate checks etc. are performed. This default behavior is designed to result in standard fully hydrogen-complete structures. If this flag is set, this step is omitted. This can for example be useful to avoid the addition of hydrogens to atoms with different default hydrogen addition characteristics if formal atomic charges have been moved. This option does apply to the input structure(s) of this command but only the first structure duplicate entered into the processing queue.
- *preservecharges*
  If set, charges are not modified after a transform is matched. By default, the charge of matched atoms is set to the charge of the matching atom in the transform template, as long as the atom has sufficient free electrons to allow the charge change. Atoms which are newly introduced by the transform always bear the charge specified in the transform description. This flag does not influence the match process - charges specified in the transform may still be used for selecting specific atoms via the checkcharges flag*preservestereo*
  If set, atom and bond stereochemistry are not changed on matched atoms and bonds. By default, changes do occur - changed atoms or bonds have their stereochemistry reset if the transform pattern does not contain stereochemistry, or set to a specific stereochemistry if it does. If only the right side of a transform contains stereochemical descriptors, the stereochemistry of the transformed product is set to that of the template (for example, a *cis* double bond). If both the left and right side of a transform contain stereochemistry, the chemistry at the transform product is inverted or retained, depending on the stereochemistry change in the transform. Having stereochemistry only on the left side is possible, and potentially useful for selecting specific enantiomers or diastereomers via the checkstereo flag, but results in a reset (if this flag is not set) or retained (if this flag is set) stereochemistry in the transformed ensemble.
- *preservecoordinates*
  If this flag is set, 2D and 3D coordinates of the transform ensemble are retained. Newly added atoms are set to a magic coordinates value. By default, a successful transformation invalidates 2D and 3D coordinates, as well as all property data dependent on these.
- *preservestereo*
  If the flag is set, all stereo descriptors are retained. By default, stereo centers and bonds matched by the pattern are reset, or inherit their new value from the pattern.
- *preservewedges*
  If set, the wedge status of bonds matching the transform pattern is preserved. By default, wedges involving bonds which are changed, or which connect atoms which are changed (or deleted and then re-added in other form), are reset. Note that this flag operates independently of the set of stereo flags listed above. In most cases, the desired mode of stereochemistry processing should be selected by specifying these flags, and the wedges regenerated as needed. If combined with stereochemistry changes, the use of this flag may otherwise lead to conflicting stereochemical information on the result ensembles.
- *removeh*
  If set, an attempt is made to rescue bond changes which would fail because of insufficient electrons for bond manipulations by deleting a minimum number of hydrogen atoms on the bond atoms needed for the bond creation or bond order change. Without this option, a transform like `[C:1][C:2]>>[C:1]=[C:2]` usually does work, since CACTVS is designed to work on structures with a full hydrogen set. When this flag is set, the transform succeeds if

C1 and C2 both have at least one hydrogen. Alternatively, the transform can be specified with explicit hydrogens as in `[#1][C:1][C:2][#1]>>[C:1]=[C:2]`. In that form, it always removes the hydrogens because they do not appear on the right side. This is form is slightly more complex and different from the Daylight mechanism.

- *requireheteromatch*
  If set, among the structure atoms matched by the pattern there must be at least one hetero atom in order to proceed with the transform modifications. If only carbon or hydrogen is matched, this match is ignored.
- *restricthydrogenmatch*
  If set, hydrogen matches are not permuted. This means that, for example, the first explicit hydrogen around a transform substructure atom can only match the first hydrogen around a structure atom, not all of them (for example, all 3 in a methyl structure group) in different matches. This is an optimization which is frequently useful, if the transform results are guaranteed to be identical regardless of which hydrogen atom was matched - for example, when generating tautomers. However, if extended attributes of the structure hydrogen atoms are significant, such as 3D position, charge or isotope labels, etc., setting this flag can lead to the non-generation of distinct result structures.
- *setatommatch*
  If this flag is set, the atom labels (`A_LABEL`) of the stored atoms on the left (substructure) side of the transform are stored on the transformation result ensembles as property `A_SSMATCH`. Atoms which are not matched are assigned a zero value.

  In case a transform result structure is the product of more than one transform, each transformation step adds a new property instance `A_SSMATCH`, `A_SSMATCH/2,` and so on. Pre-existing `A_SSMATCH` properties on the transform input ensemble are not deleted. If these exist, the new data is stored in the next unused property instance after the current instance with the highest slot number.
- *setatomstatus*
  If this flag is set, the status of the atom during the last transform is marked in property `A_TRANSFORM_STATUS`. Possible values are *none*: atom did not participate, *matched*: it was matched by the transform substructure, but did not change, *changed*: one or more atom attributes, including possibly the element number, we edited, *new*: the atom was added by the transform.

  In case a transform result structure is the product of more than one transform, each transformation step adds a new property instance `A_TRANSFORM_STATUS`, `A_TRANSFORM_STATUS/2,` and so on. Pre-existing `A_TRANSFORM_STATUS` properties on the transform input ensemble are not deleted. If these exist, the new data is stored in the next unused property instance after the current instance with the highest slot number.
- *setbondmatch*
  This option is very similar to *setatommatch* described above, except that matching left-side substructure bond labels `B_LABEL` are stored in property `B_SSMATCH`.
- *setbondstatus*
  This option is very similar to *setatomstatus* described above, except that bond history is stored in property `B_TRANSFORM_STATUS`.

- *setpathname*
  If this flag is set, the name (property `E_NAME`) of the result ensembles is set to display the transformation sequence the structure underwent from the input structure. The name is formatted as a **Tcl**-conforming list with one element for each transform applied. The first character of each list element is either '>' or '<' to indicate application of the transform in forward or reverse direction. It is followed by either the transform name (property `X_NAME`), if it is available, or the transform index number (starting with 0). Any initial name of the start structures of the transformation is cleared, so that the result name only contains transform path information.

The fifth, final, and again optional element of a **SMIRKS** line is the overlap mode. Again, if this parameter is omitted or supplied as an empty string, the global default from the command line is used. The overlap mode determines whether a transform substructure which consists of multiple disconnected fragments may match onto common target structure atoms or bonds. The following values are supported:

- *none*
  No overlap of the substructure fragments, neither on atoms nor on bonds. This is the default mode, and the most commonly used.
- *distinctmols*
  All disconnected fragments in the substructure must match different molecules in the target structure. This is a useful mode to prevent, for example, intra molecular reactions.
- *any*
  Any overlap of the substructure fragments is possible. This mode is rather useless for transforms.
- *nobonds*
  Atoms may overlap, but not bonds. This mode is actually highly useful in some contexts.
- *noembed*
  Atoms and bonds may overlap, but no substructure fragment may be completely embedded in the structure part matched by another fragment, meaning that at least one of any pair of matching substructure fragments must match an atom which is not matched by the other fragment.
- *distinctatoms*
  Between any pair of matched substructure fragments, both fragments must match at least one atom not matched by the other fragment.

Every **SMIRKS** line follows the outlined scheme, and all settings within that line are applicable only to the current transform scheme.

There is no general limit for the maximum number of transforms in this command. However, if transforms are combined with exclusion substructures, and these exclusion substructures are to be applied on a per-transform basis, (see below), the highest transform index for which an applicability flag can be set is 63. Every transform which is applied in bidirectional fashion, either by global configuration or transform-specific flags, is counted twice toward this limit.

All parameters after the **SMIRKS** lines list act globally. The third and optional *direction* parameter, command word number five, sets the default for the directionality of all transforms for which no local override was set in their respective **SMIRKS** lines. If this parameter is not specified, the default is *forward*.

The optional reaction mode, parameter four and command word six, does not have a counterpart in the **SMIRKS** lines. This parameter determines how the possibility of multiple matches of a transform substructure in the target molecules is handled. It can be one of these values:

- *first*
  Only the first match which is found is executed, all other possible matches are disregarded. The location of the first match should be considered random.
- *exhaustive*
  Only those transform products where all possible match sites have been processed are produced. For example, a structure with two reaction sites A and B, only the product where both A and B have been transformed is reported - provided that the initial transformation of A or B did not influence the possibility of matching the second part. So, in case of the hypothetical hydrolysis of a dihalogene compound with explicit water molecules, the fully reacted product will only be obtained if the input ensemble contained two water molecules. Otherwise, one (in case of symmetry) or both products of a single hydrolysis step are obtained. This mode operates by generating the intermediate products and re-submitting them. If these generate one or more new compounds, they are discarded from the result list. An older and still recognized name of this reaction mode is *all*.
- *singlestep*
  All matches are found and the transform executed, but the transform results are not re-submitted for matching as they are in the *exhaustive* mode. All different products which result from a single application of the transform are returned. For hypothetical example of the hydrolysis of an asymmetrical dihalogene compound, both partial hydrolysis products are generated, but not the fully hydrolyzed end product.
- *multistep*
  This mode generates all transform products by systematically applying the transforms to all structures and re-submitting the results again and again, until no new compounds are generated. In contrast to the *exhaustive* mode, intermediate products which further react are not discarded. The hypothetical example of the hydrolysis of an asymmetrical dihalogene compound yields three products - two partial hydrolysis products, and the fully hydrolyzed end product.

The default value for the reaction mode is *first*.

The next optional command parameter, the *selection mode*, (command argument five and command word seven) again has no counterpart in the **SMIRKS** line parameters. It determines the interaction of transforms of the same step number. All these transforms form a group. This parameter determines which of the transforms from the current group are executed, and in which order. The parameter can be set to one of the following values:

- *first*
  The first transform from the current group which matches is processed according to the reaction mode setting. All other transforms in the group are ignored, regardless whether they would match or not.
- *sequence*
  All transforms in the current group are applied once in the order they are specified, with the current reaction mode. Each transform is applied to the result ensembles of the previous transform, or the start ensemble for the first transform. All results, including those which did undergo further changes by later transforms, are returned.

- *seqendpoints*
  Similar to the *sequence* mode, but only those result ensembles which did not lead to further transformation results (either actually generated, or discarded as duplicates) are returned. Again, each transform in the sequence is only applied to the result structures of the previous transform.
- *endpoints*
  Similar to the *all* mode, but every transform is applied to all result ensembles which have accumulated before. Only those ensembles which did not yield additional, structurally distinguishable result ensembles are returned as final result.
- *all*
  All transformations are applied to all result ensembles. This process is repeated until no additional, structurally distinguishable result ensembles are generated[1]. The full set of result ensembles is returned.
- *newseqendpoints*
  This mode is similar to the *seqendpoints* mode. In *seqendpoints* mode, if a transform does not match any of the current input structures, an empty set is passed on to the next transform as input data. Thus, the transforms which follow a failing transform cannot produce any results themselves and are effectively ignored. In this mode, if a transform does not yield any results on the current input set, the current input set is re-used for the next transform, so that transforms which do not match cannot interrupt the chain. If the current transform yields results, that result set is used. The final result set is filtered, as in the *endpoints* and *seqendpoints* modes, to contain only structures which did not produce any transform results themselves.
- *parallel*
  All transforms of the current group are applied, but only to the start structure set, not to any results produced by the successful application of any previous transform.

The default selection mode is *first*.

The next and again optional *flags* parameter (command argument six, command word eight) defines the default for those transforms which do not possess an override flag set in their **SMIRKS** line. Note that if a flag set is specified on a **SMIRKS** line it completely replaces the default flag set. It does not simply add or bit-or more flags compared to the global setting. The default flag set is empty.

Similarly, the *overlap mode* parameter (command argument seven, command word nine) sets the default for handling potential overlap when matching disconnected transform fragments onto the structure to be transformed. The default setting is *none*, disallowing any fragment overlap. If the transforms only consists of a single fragment in the applicable direction(s), there is no effect of this parameter.

The *excludesslist* parameter (command parameter eight, command word ten) again has a potentially complex internal structure. It defines exclusion fragments. An exclusion fragment blocks all sections of the target structure from matching any transform substructure, either by preventing the match of transform atoms (the default) or transform bonds. This is a useful feature for example to easily prevent amide groups from matching amino group transforms. The default exclusion

1. Do not use this mode with transforms which add a group which is again matchable by the transform - you will face a runaway polymerization-style reaction!

substructure list is empty. The parameter is a list. Every list element can be a simple structure identifier, or a list of a structure identifier and a transform index list.

Structure identifiers recognized by this command are:

- ensemble handles
  This selects the complete parameter ensemble as exclusion substructure.
- lists of an ensemble handle and a molecule label
  This selects a specific molecule from the ensemble as exclusion substructure,
- **SMARTS** strings
  The **SMARTS** string is temporarily decoded and used like an ensemble handle. The transient ensemble is automatically destroyed when the **`ens transform`** command has finished.

If the exclusion substructure identifier is not associated with a transform index list, the substructure applies to all transforms. The optional transform index list consists of an arbitrary number of transform indices in the range 0...63. If a transform index list is supplied, the exclusion substructure applies only to the listed transforms. Note that it is not possible to set individual exclusion indices for transforms beyond the 64th, even though it is allowable to use any number of transforms in the transform list. All ensembles, including intermediate result ensembles, are checked against all applicable exclusion structures immediately before the application of a transform is attempted.

The exclusion substructure specification list may be prepended by a magical list element with value (*marked*)*atoms*, (*marked*)*bonds, unmarkedatoms* or *unmarkedbonds*. These control the mechanism how matched substructures are marked in the transform source structure. The default mode is *atoms*, where excluded atoms are prevented from matching transform pattern atoms. The *bonds* mode switches this to preventing a bond match. The difference is that in *bonds* mode, transform pattern atoms can still overlap, by a single atom, excluded regions, but not change bonds therein, while in *atoms* mode absolutely no atom or bond overlap between excluded regions and transform patterns is allowed. The *unmarked* variants operated with a reversed exclusion set - i.e. atoms or bonds which are *not* matched are excluded from the structure region eligible for transform application.

In case the exclusion mode is (*marked*)*atoms or unmarkedatoms*, an atom identifier, i.e. any notation which is supported to identify an atom in the **`atom`** command, may also be used in addition to the three substructure specification styles listed above to directly exclude a single atom from matching by all transforms. In (*marked*)*bonds* or *unmarkedbonds* marker mode, bond identifications in the same style as supported by the **`bond`** command, such as bond labels or bond atom label pairs, are similarly allowed as additional direct bond exclusion specifications, and these again apply to all transforms.

Exclusion markings, once set for the input structure, are inherited by newly generated result structures, so that the protection remains active even for structures undergoing sequences of transformations.

The related **`dataset transform`** command does not support direct atom or bond exclusion marking, even if the dataset only contains a single structure.

An example for an exclusion list:

```
ens transform $eh $tlist ... [list „atoms‟ {C(=O)[NH2]} {{C[NH]C} {0 1}} 1]
```

This exclusion set protects amide groups (the first substructure) from all transforms, secondary amines including their immediate carbon neighbor atoms from the first two transforms in the set (index 0 and 1, the transform set is specified in the **tlist** variable), and the single atom with label *1* in the input ensemble. The exclusion marker mode is explicitly spelled out as *atoms* in first exclusion list element, which however is already the default.

Another example:

```
ens transform $eh $tlist ... [list „unmarkedatoms" {*}$statoms]
```

This transform only operates on the atoms of which the labels or other identifiers are included in the list in variable **statoms**. All other parts of the structure are excluded and cannot participate in the transform.

The next optional global command parameter (parameter nine, command word eleven) is the maximum number of result ensembles to generate. The input ensemble is not counted. As soon as the maximum is reached, the command finishes and returns the result ensembles which were generated so far. If the maximum number of results is set to a negative number (the default), no limit applies. If it is set to zero, the transform command is effectively disabled. The global control variable *::cactvs(setsize_exceeded)* is set to 1 if the specified maximum number of result ensembles was going to be exceeded. At the beginning of the execution of the **ens transform** command, this control variable is reset to zero. The limit applies to the total of generated unique structures, which is not necessarily the same as the number of output structures in case the processing mode dictates that they are processed further and not included as intermediates in the result set. In the special case of exhaustive transform application, the parameter limits the size of the intermediate result set after each pass, not the overall total of unique structures.

The timeout parameter (command parameter ten, command word twelve) can be used to set a time limit in seconds for the command execution. If this parameter is set to 0 or a negative number, no timeout applies. This is the default. Otherwise, the generation of result ensembles is stopped after the specified time, and the command returns with the results generated so far. The global control variable *::cactvs(interrupted)* is set to 1 if a timeout occurs. It is reset to 0 at the beginning of the execution of the command.

The next optional parameter (command parameter eleven, command word thirteen) can be used to limit the number of transforms applied to the starting structure and intermediate structures. If this parameter is not specified, or specified as an empty string or a negative value, no limit is imposed. If this parameter or the timeout option is used, the result set may become dependent on the atom and bond order of the input structure because the traversed part of the possible transform match space is different and might yield different and/or a different number of results when the timeout or application count restriction is triggered.

The second last optional parameter (command parameter twelve, command word fourteen) is an iteration count. Its default value is one, meaning that the whole transformation process is only executed once. If set to a larger value, the transformation routine calls itself recursively. This is equivalent to first running **ens transform** with a start structure, and then repeatedly execute **dataset transform** commands for the second and later iterations with the last result set. All limits and other control parameters are passed in the original configuration, and apply only to the next iteration, not globally over the sum of all transform cycles. By default, the result set of this mode is what the last iteration produced, but this can be changed to the union of all iteration results by the *keepiterationintermediates* flag. Uniqueness checking of result structures is applied to the full return set. If the parameter is set to zero or a negative value, no transformations are executed. If the *setpathname* flag is set, it is automatically switched to *appendpathname* for the second and later cycles, so that the name mirrors the full transformation history and is not reset in each cycle.

The final optional parameter is an array variable name. If it is specified, various statistics about the transform application are collected and stored in that array. Some important array elements are:

• *patternmatches*	The total number of transform pattern substructure matches, on both sides of the transform if the transform settings allow this
• *leftpatternmatches*	The total number of left-side transform pattern substructure matches
• *rightpatternmatches*	The total number of right-side transform pattern substructure matches
• *leftpatternmatches$n*	The number of left-side pattern matches for transform pattern *n*, beginning with index 0
• *rightpatternmatches$n*	The number of right-side pattern matches for transform pattern *n*, beginning with index 0
• *datafailures*	The number of aborted transform applications because property data could not be computed on the transform result structures
• *applicationfailures*	The total count of failures to apply the transform instructions of a matched pattern, for example because of bad electron counts
• *applicationsuccesses*	The total count of successful applications of transforms, before the duplicate check
• *duplicaterejections*	The number of successfully transformed structures which were not added to the result set because they were a duplicate of an already-registered structure
• *duplicateaccepts*	The number of transform result structures which passed the duplicate result structure check

Example:

```
set t1 {{[O,S;X1:1]=[C:2x1][C:3X4][#1:4]>>[#1:4][O,S;X2:1][C:2x1]=[C:3]
enol/thioenol}}
set elist [ens transform $eh [list $t1] bidirectional multistep all preservecharges
none]
```

This example is part of a tautomer generator. The full standard generator in the toolkit uses a lengthy list of transform schemes and not just the one sample keto/enol schema displayed here. Because the operation is bidirectional, the transform transforms ketones into enols, and vice versa. If more than one interchangeable group exists, all intermediate structures are generated (*multistep* reaction mode). All results are retained (*all* selection mode), and all intermediate structures are again subjected to all transforms (this does not have any effect with a single transform, but the real application uses a set of transforms). Finally, charges should not be changed (*preservecharges* flags), and fragment overlap is not allowed (*none* overlap mode) - this again is without effect in this sample transform, because it does not consist of disconnected fragments on either side.

Multiple structures may be jointly transformed in a single command by means of the very similar **`dataset transform`** command.

ens translate

```
ens translate ehandle pt1 ?pt2? ?property?
e.translate(point1=,?point2=?,?coordinateproperty=?)
```

Move the atoms of the ensemble by modifying their 3D coordinates in property `A_XYZ`, or a custom atomic float vector coordinate property. This command requires atomic 3D coordinates and will attempt to compute them if they are not yet present. If no 3D atomic coordinates can be generated, the command fails with an error.

The first argument is interpreted as a 3D vector if this is the only coordinate argument. All atoms with valid 3D coordinates are moved according to the vector coordinates. In case a second argument is supplied, both arguments are interpreted as points in 3D space. The ensemble atoms are moved according to the difference vector between the second and the first point.

This operation triggers a *3dglop* property invalidation event.

The command returns the original ensemble handle or reference.

Examples:

```
ens translate $eh {0 0 1}
ens translate $eh [atom get $eh $a1 A_XYZ] [atom get $eh $a2 A_XYZ]
```

ens trim

```
ens trim ehandle ?propertylist?
e.trim(?properties=?)
```

Reduce the information content of a structure to a standard minimum set and discard any additional information. This process minimizes the storage requirements of the ensemble. The properties of the internally defined minimum set are computed if required. The retained property set is designed to support a faithful representation of connectivity including bond and atom labels and types as well as formal charges, stereochemistry, isotopes, 2D and 3D coordinates, but not of auxiliary additional attributes of atoms, bonds or other minor objects.

The optional fourth argument is a list of properties which should be retained in addition to the standard set. If any of these are not present on the ensemble to be trimmed, they are silently ignored and no attempt is made to compute them. Specifying properties of the standard retention set in this list is allowed but has no additional effect.

The return value of the command is a list of the remaining properties of the ensemble.

Example:

```
ens trim $ehandle {E_GIF E_SMILES}
```

ens uncharge

```
ens uncharge ehandle ?filterset? ?flags?
e.uncharge(?filters=?,?flags=?)
```

Attempt to remove charges on atoms in a chemically sensible way. Charge removal by default happens via addition or removal of protons. In cases where this does not make chemical sense, a direct charge manipulation may be performed instead. Charged metal ions and other charged species without an obvious method for neutralization remain unchanged.

By default all atoms are processed, but the set of processed atoms can be limited by specifying a filter collection. Additional conditions on processed atoms can be set via the flag argument, which accepts the same values as **ens hadd**. Please refer to that command for a list and explanation of these flags.

The command returns the number of atoms which were neutralized.

Example:

```
ens uncharge [ens create {[NH3+]CC(=O)[O-]}]
```

This sample line removes a proton from the charged amino group and add a proton to the charged carboxyl group of the initial glycine zwitterion. The returned result value is 2. In this example the total hydrogen count has not changed. In case of an unbalanced set of positive and negative, modified charged centers this is usually not the case.

ens unlock

```
ens unlock ehandle propertylist/objclass/all
e.unlock(property=)
```

Unlock property data for the ensemble, meaning that they are again under the control of the standard data consistency manager.

The property data to unlock can be selected by providing a list of the following identifiers:

- Property names or references
  Valid property instances on the ensemble, or ensemble minor objects are unlocked. Non-existent data is silently ignored. It is not possible to unlock individual property fields.
- *all*
  All valid ensemble or ensemble minor object properties are unlocked. Ensemble properties and ensemble minor object properties are not affected.
- *ens,atom,bond...*
  These are object class identifiers. All property data which is controlled by the ensemble major object and attached to the specified object class is unlocked.

Property data locks are obtained by the **ens lock** command.

The return value is the original ensemble handle or reference.

Example:

```
set eh [ens create CCC]
ens lock $eh A_SYMBOL 1
ens purge $eh A_ELEMENT
atom set $eh 1 A_query(dsearch) 3
ens unlock $eh A_SYMBOL
```

In this example, an ensemble is created, and the atom symbol information is locked. Next, the element number property is deleted, and a query attribute is set. Finally, the lock is released. Had the element symbol information not been locked, the ensemble would have become unusable due to an overzealous data consistency manager. Setting query information in property `A_QUERY` *can* have an influence on the atom symbol. So the default action of invalidating `A_SYMBOL` when manipulating `A_QUERY` is correct. However, in case there is no element information `A_ELEMENT`, and no atom symbol information `A_SYMBOL`, the element information is completely lost, and the ensemble becomes unusable. So in this case, locking `A_SYMBOL` (or alternatively `A_ELEMENT`) is required to avoid unexpected side effects of structure editing.

ens unpack

```
ens unpack packstring ?compressionlib?
Ens.Unpack(data=,?compressionlib=?)
```

Unpack a base64-encoded serialized object string which was created by an **`ens pack`** command. The return value of this function is the handle of the newly created ensemble object, which is an exact duplicate of the packed original ensemble.

Packed ensembles may also be unpacked by the **`ens create`** command.

The default compression library is *zlib*. For more options, see **`ens pack`**.

Example:

```
set packdata [ens pack [ens create CCCl]]
set ehandle [ens unpack $packdata]
```

ens valencecheck

```
ens valencecheck ehandle ?failedatomvariable? ?nitrogenmode?
e.valencecheck(?variable=?,?nitrogenmode=?)
```

Perform a valence check on the ensemble, comparing the current bonding situation at all atoms to the list of element-specific valence states in the system element table. This command is intentionally quite picky, discouraging for example the use of pentavalent nitrogen by default. For the calculation of valence, only bonds of type *normal* (valence bonds) are taken into account. *Complex* bonds and pseudo bond types thus do not interfere in the calculation. Some more exotic metal atoms with many different valence states, or few well-defined covalent compounds, such as *vanadium* or *rhodium*, always pass.

The handling of nitrogen in pentavalent or ionic form can be controlled by setting the optional *nitrogenmode* argument, or modifying the global **`::cactvs(nitrogen_valence_check)`** variable.Possible values are *xionic*, *ionic* (the default), *asis*, *pentavalent* and *xpentavalent*. These are the same values as with the **`ens nitrostyle`** command - please refer to that command for more information. In *asis* mode, both ionic and pentavalent forms pass.

The return value of this command is the number of atoms which failed the valence check. If the optional *failedatomvariable* argument is specified as non-empty string, it is the name of a variable which receives a list of the atom labels which failed the check, or is set to an empty list in case no problems were found.

Note that this command assumes that all hydrogen atoms are in place. Processing of structures with implicit hydrogen atoms is not supported.

`mol valcheck` is a short command alias.

Example:

```
ens valencecheck [ens create {CN(=O)=O.C[N+](=O)[O-]}] badatoms
```

This sample command checks the valence situation of ***nitromethane*** in two encoding formats. The first molecule, using a pentavalent nitrogen encoding, is responsible for the result value 1, indicating one failed atom, and the variable *badatoms* is set to 2, the label of the pentavalent nitrogen atom. The second molecule passes the check and reports no additional problems.

`ens valcheck` is a short alias.

ens valid

```
ens valid ehandle propertylist
e.valid(property/propertysequence)
```

Returns a list of boolean values indicating whether values for the named properties are currently set for the ensemble. No attempt at computation is made. For **Python**, where single-item lists are syntactically not the same as a single value, the return value is a single boolean if the argument was a string or a property reference, and only a single property was decoded.

Example:

```
ens valid $xhandle X_IDENT
```

reports whether the ensemble has a standard ID (has a valid `E_IDENT` property) or not.

`ens has` is an alias to this command.

ens vector

```
ens vector ehandle property vectorname ?invert? ?integrate?
```

Map ensemble property data to a **Blt** library vector object. Please refer to the **Blt** manual pages for more information on these. **Blt** vector objects are very useful, for example, for the efficient set-up of GUI graphing widgets which are provided by the **Blt Tk** extension. This command automatically attempts to load the **Blt** Tcl module if necessary. If that fails, an error results.

The vectorized property data must be of a vector type, and the element type of the vector must either be a simple numeric type, or a bit for bitvectors, or a floating-point pair. It is possible to address a property field, for example the X/Y data points of a spectrum which are typically stored as a field in a complex compound property.

If the *invert* flag is set, the stored **Blt** vector object values are set to 1.0 minus the property data value. By default, this flag is not active. If the *integrate* flag is set, the **Blt** vector object element values are set to the sum of all preceding property data values. This flag is also disabled by default.

If the property data type is a float pair vector, two vector objects are created in the **Blt** namespace, with suffixes *_X* and *_Y*. For simple vector types, the vector name is used directly. It is possible to overwrite existing **Blt** vectors of the same name with this command.

The return value of the command is a list of the generated name of the vector, followed by the minimum and maximum data values in that vector object. These may the different from the ensemble property data values because of the application of the *invert* or *integrate* flags.For float pair vectors, the same information is repeated for the second vector object.

The command is not supported in the **Python** interface.

ens verify

```
ens verify ehandle property
e.verify(property)
```

Verify the values of the specified property on the ensemble. The property data must be valid, and of an ensemble or ensemble minor object property. If the data can be found, it is checked against all constraints defined for the property, and, if such a function has been defined, is tested with the value verification function of the property.

If all tests are passed, the boolean return value is boolean 1, 0 if the data could be found but fails the tests, and an error condition otherwise.

ens weed

```
ens weed ehandle keywords
e.weed(keywordsequence)
e.weed(?keyword?,...)
```

This command performs a number of common clean-up and standardization operations on the ensemble, which are especially useful in the context of processing **PDB** files. The ensemble is potentially modified, but keeps its handle or reference, which is returned as command result. In addition, properties `A_XYZ` and `A_RESIDUE`, which are normally susceptible to bond manipulations, are locked and retained.

The *keywords* argument selects the desired set of operations. Most of the keywords are single words, but the *minsize* and *maxsize* as well as the *minaminoacids* and *maxaminoacids* keywords take an additional integer number as argument. The following operations are currently supported:

- *carbonless*
  Remove all molecules/fragments which do not contain carbon.
- *disulphides*
  Split and hydrogenate all disulfide bridges. This operation can change the molecule and ring set.
- *duplicates*
  Remove all molecules/fragments which are duplicates (taking isotope labels and stereochemistry into account) of another molecule in the ensemble. Only a single instance of any duplicate molecule is retained. Internally, this is a check on property `M_HASHISY`.
- *hydrogenless*
  Remove all molecules which do not contain hydrogen.
- *inorganic*
  Remove all inorganic molecules.
- *ligands*
  Remove all molecules which do not consist exclusively of linked standard amino acids. This flag is complementary to *proteins*.
- *maxaminoacids n*
  Discard all molecules from ensemble which consist only of linked standard amino acids and contain more than the specified number of them. This operation requires an additional integer after the keyword.

- *maxsize n*
  Discard all molecules from ensemble which have more than the specified number of atoms. This operation requires an additional integer after the keyword.
- *metalatoms*
  Remove all metal atoms from the ensemble. This operation can change the molecule and ring set.
- *metalions*
  Remove all molecules which are unbonded metal atoms. Bonded metal atoms are not affected.
- *metaloxygenbonds*
  Remove all bonds between metal atoms and oxygen atoms. This operation can change the molecule and ring set.
- *minaminoacids n*
  Discard all molecules from ensemble which consist only of linked standard amino acids and contain less than the specified number of them. This operation requires an additional integer after the keyword
- *minsize n*
  Discard all molecules from ensemble which have less than the specified number of atoms. This operation requires an additional integer after the keyword.
- *proteins*
  Discard all molecules which only consist of linked standard amino acids. This is a shortcut for *minaminoacids 0*.
- *proteinhetatmbonds*
  Discard all bonds between the protein core and heterogens, i.e. all bonds where the property field `A_RESIDUE(HETATOM)` is different among the involved bond atoms. This operation can change the molecule and ring set.
- *proteinspecialbonds*
  Discard all special bonds (i.e. complex bonds, link bonds, etc.) where at least one atom is from the protein, i.e. was encoded with an **`ATOM`** line in a **PDB** file, not **`HETATM`**. This operation can change the molecule and ring set.
- *specialbonds*
  Delete all bonds which are not VB bonds. This operation can change the molecule and ring set.
- *water*
  Discard water molecules, i.e. all molecules which consist of one oxygen atom, any number of hydrogen atoms, and no other element.

The order of the keywords is not important. The sequence of operations is always

metalatoms > specialbonds > proteinspecialbonds,proteinhetatmbonds > metaloxygenbonds > disulphides > carbonless,hydrogenless,inorganic,maxsize,metalions,minsize,water > maxaminoacids,minaminoacids > duplicates

Applied operations which potentially change the set of molecules and rings trigger an automatic re-evaluation of this data after the operation block has been executed.

Example:

The code below is part of a reliable **PDB** ligand extractor.

```
ens weed $eh {metaloxygenbonds water proteinspecialbonds duplicates minsize 10 \
maxsize 300 maxaminoacids 6 disulfides}
if {[ens get $eh E_NATOMS]==0} {
# try again with additional bond cut step. Cannot do this by default, because
# there are plenty of ligands with embedded amino acid parts
# that are encoded as ATOM lines. PDB files suck.
   molfile backspace $fh
   set eh [molfile read $fh]
   ens weed $eh {metaloxygenbonds water proteinspecialbonds proteinhetatmbonds \
duplicates minsize 10 maxsize 300 maxaminoacids 6 disulfides}
}
```

ens xhandle

`ens xhandle` *`ehandle`*

Return the remote handle of the ensemble if it was exported and is currently under the control of a live-linked application. In case the ensemble is not exported, an error results.

This command is not supported in the **Python** interface.

The *group* Command

The *group* command is the generic command used to manipulate groups. The syntax of this command follows the standard schema of *command/subcommand/majorhandle/minorlabel.*

Pseudo group labels *first*, *last* and *random* are special values, which select the first group in the group list, the last, or a random group.

Example:

```
group get $ehandle 1 G_SIZE
```

This is the list of officially supported subcommands:

group add

```
group add ehandle label object/objectlist...
g.add(?object/objectsequence?,...),
```

Add more atoms or groups as members to an existing group. A group cannot be added to itself, and the formation of cyclic dependencies is illegal. It is however possible to add an atom or group more than once to a group, and an atom or a group may be a member of an arbitrary number of groups.

Adding objects to a group triggers a *groupchange* property invalidation event and may thus have an influence on the validity of chemical object data.

The use of an empty object list is possible and does not change the group, nor is an invalidation event issued.

The object list syntax is the same as in the **`group create`** command.

The command returns the original group label or reference.

Examples:

```
group add $ehandle $glabel 1
group add $ehandle $glabel [list "group" [group create $ehandle [list 5 7]]]
```

The first sample line simply adds the atom with label 1 to the group. The second line adds a newly created group with atoms 5 and 7 to the existing group as a recursive group element.

group append

```
group append ehandle label ?property value?...
g.append({?property:value,?...})
g.append(?property,value,?...)
```

Standard data manipulation command for appending property data. It is explained in more detail in the section about setting property data.

The command returns the first data value.

Example:

```
group append $ehandle 1 G_NAME "_linker"
```

group atoms

```
group atoms ehandle label ?filterset? ?filtermode?
```

```
g.atoms(?filters=?,?mode=?)
```

List the labels or references of all atoms which are in the group. There are two different modes of operation, depending on whether the group contains at least one atom as member object.

If there is a member atom: Group member objects which are not atoms, such as bonds or recursive groups, are omitted from output, as are atoms which are only indirectly a group member via a recursive group.

Without atoms in the group, atoms which are components of the group objects are listed, e.g. the atoms of bonds that are in the group.

In other respects, this is a standard cross-referencing command. The usage of the *filterset* and *filtermode* parameter is explained in the section about object cross-references.

Example:

```
set gh [group create $ehandle [list 1 2 3]]
group atoms $ehandle $gh carbon
```

gets the labels of the carbon atoms in the group.

```
set gh [group create $ehandle [list [list „bond" 1]]]
group atoms $ehandle $gh
```

while this command on a group which only contains bonds, but no atoms, reports the atom labels of the bond in the group.

group bonds

```
group bonds ehandle label ?filterset? ?filtermode?
g.bonds(?filters=?,?mode=?)
```

Retrieve the labels or references of bonds which are associated with a group. There are two different modes of operation, depending on whether the group contains at least one bond as member object.

If there is a member bond: Group member objects which are not bonds, such as atoms or recursive groups, are omitted from output, as are bonds which are only indirectly a group member via a recursive group.

Otherwise, a bond is considered to be associated with a group if all atoms of the bond are group members. All bond atoms must be in the same group object, i.e. indirect memberships via recursive groups are ignored. Bonds are not associated with a group if only some of their atoms are members of the group.

In other respects, this is a standard cross-referencing command. The usage of the *filterset* and *filtermode* parameter is explained in the section about object cross-references.

Example:

```
set gh [group create $ehandle [list 1 2 3]]
group bonds $ehandle $gh {1 doublebond triplebond}
```

gets the bond labels of all double and triple bonds between the group atoms.

```
set gh [group create $ehandle [list [list „bond" 1]]]
group bonds $ehandle $gh
```

while this command directly lists the bond in the group.

group create

```
group create ehandle objectlist...
Group(?objectref?,...)
Group.Create(?object?ref,...)
Group(eref,?objectref/objectrefsequence/objectlabel?,...)
Group.Create(eref,?objectref/objectrefsequence/objectlabel?,...)
```

Create a new group containing atoms or other minor objects, including other groups as member elements.

The object list parameter is a list of object identifiers. An object identifier is either a single-element simple identifier, in which case it is interpreted as an atom identifier (usually a label, but all other identifiers are possible), or a two-element list. If the second form is used, the list must consist of an object class name, followed by an object identifier (usually a label, but all types of minor object identifiers are possible).

Specifying a member object which cannot be resolved produces an error. However, it is no error for an atom or a group to be listed more than once as a member of a group, nor is there any restriction of how many groups an atom or other minor object can be a member. However, circular relationships are illegal, and a group cannot be a member of itself. Duplicate objects in a group are allowed and not filtered when a group is set up.

Creating a new group triggers the *group* and *groupchange* invalidation events and may thus influence the validity of chemical object data.

The creation of empty groups by supplying an empty object list is possible.

The return value of this function is the label of the new group.

Examples:

```
set g1 [group create $ehandle {1 2 3}]
set g2 [group create $ehandle [list [list "group" $g1] 4 [list "atom" #5]]]
```

The first line creates a simple group with atoms 1, 2 and 3. The second line builds a recursive group which contains the first group (identified as a group reference by prefixing its label with a *group* object class name), the atom with label 4, and the atom with index 5 (which could have any label).

group defined

```
group defined ehandle label property
g.defined(property)
```

This command checks whether a property is defined for the group. This is explained in more detail in the section about property validity checking. Note that this is *not* a check for the presence of property data! The **`ens valid`** command is used for this purpose.

Example:

```
group defined $ehandle 1 G_XYZ
```

checks whether group 1 is of a type for which `G_XYZ` is defined.

group delete

```
group delete ehandle ?label?...
group delete ehandle all
```

```
g.delete()
Group.Delete(eref,"all")
Group.Delete(gref,...)
Group.Delete(eref,?glabel/gref/grefsequence?,...)
```

Delete a set of groups. Groups are either identified by a standard group identifier (usually a label), or the reserved word *all*.

If a deleted group contains as a member another group, that group is also deleted in a recursive fashion. If this behavior is not wanted, recursive groups should be explicitly unlinked from their base groups by means of the **`group remove`** command.

Deleting a group triggers the *group* and *groupchange* invalidation events and may thus influence the validity of chemical object data. If an empty object list is used, the command does nothing, and no invalidation event is generated.

This command returns the number of deleted groups on the first level, i.e. recursive group deletions are not counted.

Examples:

```
group delete $ehandle all
group delete $ehandle [ens groups $ehandle xatom]
```

The first example deletes all groups in the ensemble. The second example deletes all those groups which contain one or more hetero atoms as members.

group dget

```
group dget ehandle label propertylist ?filterset? ?parameterdict?
g.dget(property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **`group get`** command. The difference between **`group get`** and **`group dget`** is that the latter does not attempt computation of property data, but rather initializes the property values to the default and return that default if the data is not yet available. For data already present, **`group get`** and **`group dget`** are equivalent.

group dup

```
group dup ehandle label ?datasethandle? ?position?
g.dup(?target=?,?position=?)
```

Duplicate the atoms and bonds of a group into a new ensemble. The function returns the new ensemble handle or reference.

By default, the new ensemble is appended to the same dataset as the original ensemble, or placed outside of any dataset if the input ensemble was not a dataset member. If the optional target dataset parameter is specified, the duplicate is directly moved to that dataset. If an empty string is passed (or **`None`** for **PYTHON**), the duplicate is not made a dataset member, even if the input ensemble is in a dataset.

Example:

```
group dup $ehandle 1 [dataset create]
```

duplicates the group with label one and move the new single-molecule ensemble into a newly created dataset.

group ens

```
g.ens()
```

Python-only method to get the ensemble reference from a group reference.

group exists

```
group exists ehandle label ?filterlist?
g.exists(?filters=?)
Group.Exists(eref,label,?filters=?)
```

Check whether this group exists. Optionally, a filter list can be supplied to check for the presence of specific features. The command returns boolean 0 if the group does not exist, or fails the filter, and 1 in case of successful testing.

Example:

```
group exists $ehandle 99
```

group expr

```
group expr ehandle label expression
g.expr(expression)
```

Compute a standard **SQL**-style property expression for the group. This is explained in detail in the chapter on property expressions.

group fill

```
group fill ehandle label ?property value?...
g.fill({property:value,...})
g.fill(?property,value?,...)
```

Standard data manipulation command for setting data, ignoring possible mismatches between the lengths of the lists of objects associated with the property and the value list. It is explained in more detail in the section about setting property data.

The command returns the first fill value.

Example:

```
group fill $ehandle 1 B_COLOR red
```

sets the color of the first bond group 1 contains or is associated with to *red*.

group filter

```
group filter ehandle label filterlist
g.filter(filters)
```

Check whether a group passes a filter list. The return value is boolean 1 for success and 0 for failure.

Example:

```
group filter $ehandle 1 [list carbon doublebond]
```

checks whether the group contains one or more carbon atoms and one or more double bonds. The double bond does not need to be with a carbon atom.

group formulamatch

```
group formulamatch ehandle label formula_expression ?other_elements?
g.formulamatch(query=,?other_elements=?)
```

Match the group against a formula expression. Its syntax is the same as in formula queries in **`molfile scan`** and other scan commands.

There are several methods to specify whether any elements not mentioned in the formula expression may or must be present. If the *other_elements* flag is used, it has the highest priority. If may be set to 0 (no other elements allowed), 1 (allowed) or 2 (required), and if it is set, any prefix in the formula expression is ignored. If it is not used, a prefix in the formula expression may be used to control the matching. Supported prefixes are = (no other elements), >= (other elements allowed) and > (required). If no prefix is used, the default mode is an exact match without other elements.

The return value is the boolean match result.

Examples:

```
group formulamatch $eh 1 >C6
```

Tests whether the group contains six carbon atoms. At least one atom which is not carbon must be present.

```
group formulamatch $eh 1 C5-6(Cl+Br+I)2- 1
```

Tests whether the group has five or six carbon atoms, two ore more heavy halogens, and potentially any other elements.

group get

```
group get ehandle label propertylist ?filterset? ?parameterdict?
g.get(property=,?filters=?,?parameters=?)
g[property]
g.property
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

Examples:

```
group get $ehandle 1 {G_SIZE A_ELEMENT}
```

yields a list with two elements, consisting of the group size (count of group members) as the first element and the element numbers of all atoms in the groups as a nested list as the second result list element. If the information is not yet available, an attempt is made to compute it. If the computation fails, an error results.

```
group get $ehandle 1 B_ORDER ringbond
```

gives the bond orders of all bonds of associated with the group which are ring bonds.

For the use of the optional property parameter list argument, refer to the documentation of the **`ens get`** command.

Variants of the *group get* command are *group new, group dget, group nget, group show, group sqldget, group sqlget, group sqlnew* and *group sqlshow.*

Further examples:

```
group get $ehandle 1 E_NAME
group get $ehandle 1 A_FLAGS(boxed)
```

group group

```
group group ehandle label
Group.Ref(eref,identifer)
```

Standard cross-referencing command to obtain the label or reference of the group as stored in property `G_LABEL`. This is explained in more detail in the section about object cross-references. Note that there is also a **`group groups`** (plural *groups*) command which has a different function.

Example:

```
group group $ehandle #0
```

returns the label of the first group of the ensemble group list.

group groups

```
group groups ehandle label ?filterset? ?filtermode?
g.groups(?filters=?,?mode=?)
```

List the labels or references of all groups which are members of the group. Group member objects, such as atoms, which are not groups are omitted, as are groups which are only indirectly a group member via a recursive group.

In other respects, this is a standard cross-referencing command. The usage of the *filterset* and *filtermode* parameter is explained in the section about object cross-references. Note that there is also a **`group group`** (singular *group*) command which has a different function.

Example:

```
group groups $ehandle 1
```

gets the labels of the groups which are a (recursive) member of group 1.

group hdup

```
group hdup ehandle label ?datasethandle? ?position?
g.hdup(?target=?,?position=?)
```

This command provides the same functionality as **`group dup`**, except that it also adds a standard set of hydrogens to the new ensemble.

group index

```
group index ehandle label
g.index()
```

Get the index of the group. The index is the position in the group list of the ensemble. The first position is index 0.

Example:

```
group index $ehandle 99
```

group jget

```
group jget ehandle label propertylist ?filterset? ?parameterdict?
g.jget(property=,?filters=?,?parameters=?)
```

This is a variant of **`group get`** which returns the result data as a **JSON** formatted string instead of **Tcl** or **Python** interpreter objects.

group jnew

```
group jnew ehandle label propertylist ?filterset? ?parameterdict?
g.jnew(property=,?filters=?,?parameters=?)
```

This is a variant of **`group new`** which returns the result data as a **JSON** formatted string instead of **Tcl** or **Python** interpreter objects.

group jshow

```
group jshow ehandle label propertylist ?filterset? ?parameterdict?
g.jshow(property=,?filters=?,?parameters=?)
```

This is a variant of **`group show`** which returns the result data as a **JSON** formatted string instead of **Tcl** or **Python** interpreter objects.

group local

```
group local ehandle label propertylist ?filterset? ?parameterdict?
g.local(property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading and recalculating object data. It is explained in more detail in the section about retrieving property data.

Example:

```
group local $ehandle 1 A_LABEL_STEREO
```

Note that very few computation routines currently support the local re-computation of data - in most cases, this command falls back to a global re-computation.

group match

```
group match ehandle label ss_ehandle ?ss_label? ?matchflags? ?ignoreflags?
   ?atommatchvar? ?bondmatchvar? ?molmatchvar?
g.match(substructure=,?substructuregroup=?,?matchflags=?,?ignoreflags=?,
   ?atommatchvariable=?,?bondmatchvariable=?,?molmatchvariable=?)
```

Check whether the selected group matches a substructure. Only the first substructure group, or the group selected by the substructure label parameter, is tested. The substructure may be part of any structure ensemble, and even be in the same ensemble as the primary command group. Both the atoms in the group and the bonds between them are checked.

The precise operation of the substructure match routine can be tuned by providing a standard set of match flags and feature ignore flags. The default match flag set has set bits for the *bondorder*, *atomtree* and *bondtree* comparison features, and an empty ignore set. If a flag set is specified as an empty string, the default set is used. In order to reset a flag set, an explicit *none* value must be used.

The command returns 1 for a successful match, 0 otherwise. If an optional atom, bond, or molecule match variable is specified, it is set to a nested list of matching substructure/structure atom, bond or molecule labels. If no match can be found, the variable is set to an empty list. In case only a bond or molecule match variable is needed, an empty string can be used to skip the unused match variable argument positions.

Example:

```
set ss [ens create {c1ccccc1} smarts]
set g_contains_phenyl_ring [group match $ehandle $label $ss]
```

group mols

```
group mols ehandle label ?filterset? ?filtermode?
g.mols(?filters=?,?mode=?)
```

Standard cross-referencing command to obtain the labels or references of the molecules the atoms in the group are a member of. This is explained in more detail in the section about object cross-references.

Examples:

```
group mols $ehandle 1
group mols $ehandle 1 heterocycle
```

The first example is simple retrieval, the second line filters the molecules and lists only the labels of those molecules which contain one or more heterocycles.

group new

```
group new ehandle label propertylist ?filterset? ?parameterdict?
g.new(property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **`group get`** command. The difference between **`group get`** and **`group new`** is that the latter forces the re-computation of the property data, regardless whether it is present and valid, or not.

group nget

```
group nget ehandle label propertylist ?filterset? ?parameterdict?
g.nget(property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **`group get`** command. The difference between **`group get`** and **`group nget`** is that the latter always returns numeric data, even if symbolic names for the values are available.

group objects

```
group objects ehandle label ?filterset? ?filtermode?
g.objects(?filters=?,?mode=?)
```

This is a cross-referencing command specific to groups. The standard operation of cross-referencing commands and the use of the optional parameters are explained in the object referencing section of this manual.

The difference of this command to the **`group atoms, group bonds`** or **`group groups`** commands is that this command lists all object classes present in the group. Every listed item is output as a list with two elements - the first being the object class (*atom, bond* or *group*), the second being the object label or reference. This list is suitable for use in a **`group create`** or **`group add`** statement.

The command returns the label or reference of the new group.

Example:

```
group create $ehandle [group objects $ehandle 1]
```

This command duplicates the group with label 1.

group pis

```
group pis ehandle label ?filterset? ?filtermode?
g.pis(?filters=?,?mode=?)
```

Standard cross-referencing command to obtain the labels or references of the π systems the group is associated with. This is explained in more detail in the section about object cross-references.

Examples:

```
group pis $ehandle 1
```

π systems are a rather exotic feature and not commonly used. These are essentially descriptions of bonding interactions which use p or d orbitals, such as in standard covalent multiple bonds. A simple double bond is described with one σ system and one π system in this representation.

group ref

```
Group.Ref(eref,identifier)
```

PYTHON only method to get a group reference. See **`group group`** command.

group remove

```
group remove ehandle label objectlist...
g.remove(?object/objectsequence?,...)
```

Remove group items from a group. The removed objects are not deleted from the ensemble, they simply are no longer a group member. The syntax of the object list is the same as in the **`group add`** and **`group create`** commands. The groups the objects are removed from also remain in existence.

Removing an object from a group triggers a *groupchange* property invalidation event and may thus have an influence on the validity of chemical object data.

The command returns the number of removed group elements.

Examples:

```
group remove $ehandle 1 [group atoms $ehandle 1 hydrogen]
```

This command removes all hydrogen atoms from group 1.

group replicate

```
group replicate ehandle label ?count? ?creategroups?
g.replicate(?count=?,?creategroups=?)
```

Add copies of a group to the current ensemble as a new molecule/fragment. All atoms of the group are replicated, and the bonds which are either explicitly part of the group, or, if these are not set, all bonds between the group atoms are also created in the new fragment. The default duplication count is one.

If the *creategroups* boolean flag is set, the duplicated atoms are also registered as a new group.

If new groups are created, the return value is a list of the handles or references of the new groups. If no new groups are created, the return value is the handle or reference of the argument ensemble.

group rings

```
group rings ehandle label ?filterset? ?filtermode?
g.rings(?filters=?,?mode=?)
```

Standard cross-referencing command to obtain the labels or references of the rings the group is associated with. This is explained in more detail in the section about object cross-references. Rings which only partially overlap with the group are included.

Examples:

```
group rings $ehandle 1
group rings $ehandle 1 [list heterocycle aroring]
```

The first example returns the labels of all rings the group overlaps with. If the group does not overlap with any ring, an empty list is returned. Only labels of rings in the SSSR or ESSSR set are returned, even if the currently computed ring set is larger. The second example filters the rings - only heteroaromatic rings are reported.

group ringsystems

```
group ringsystems ehandle label ?filterset? ?filtermode?
g.ringsystems(?filters=?,?mode=?)
```

Standard cross-referencing command to obtain the labels or references of the ring systems the group is associated with. This is explained in more detail in the section about object cross-references. Ring system which only partially overlap with the selected group are listed.

Examples:

```
group ringsystems $ehandle 1
group ringsystems $ehandle 1 [list heterocycle aroring]
```

The first example returns the labels of all ring systems the group is associated with. If the group does not overlap with any ringsystem, an empty list is returned. The second example filters the ring systems - a ring system label is added to the output list only if that ring system contains one or more hetero aromats.

group set

```
group set ehandle label ?property value?...
g.set(?property,value?,...)
g.set({property:value,...})
g.property = value
g[property] = value
```

Standard data manipulation command for setting property data. It is explained in more detail in the section about setting property data.

The command returns the first data value.

Example:

```
group set $ehandle 1 G_CONSTRAINT [list "distance" [list 3.0 4.0]]
```

group show

```
group show ehandle label propertylist ?filterset? ?parameterdict?
g.show(property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the *group get* command. The difference between **`group get`** and **`group show`** is that the latter does not attempt computation of property data, but raises an error if the data is not present and valid. For data already present, **`group get`** and **`group show`** are equivalent.

group sigmas

```
group sigmas ehandle label ?filterset? ?filtermode?
g.sigmas(?filters=?,?mode=?)
```

Standard cross-referencing command to obtain the labels or references of the σ systems the group is associated with. This is explained in more detail in the section about object cross-references. An association is assumed if any atoms of the σ system is a group member. Recursive groups are not searched.

Examples:

```
group sigmas $ehandle 1
```

σ systems are a rather exotic feature and not commonly used. These are essentially descriptions of bonding interactions which use s orbitals, such as normal, covalent single bonds, or the central bond in multiple bonds. A simple double bond is described with one σ system and one π system in this representation.

group sqldget

```
group sqldget ehandle label propertylist ?filterset? ?parameterdict?
g.sqldget(property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **`group get`** command. The differences between **`group get`** and **`group sqldget`** are that the latter does not attempt computation of property data, but initializes the property value to the default and returns that default, if the data is not present and valid; and that the **SQL** command variant formats the data as **SQL** values rather than for **TCL** or **PYTHON** script processing.

group sqlget

```
group sqlget ehandle label propertylist ?filterset? ?parameterdict?
g.sqlget(property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **`group get`** command. The difference between **`group get`** and **`group sqlget`** is that the **SQL** command variant formats the data as **SQL** values rather than for **TCL** or **PYTHON** script processing.

group sqlnew

```
group sqlnew ehandle label propertylist ?filterset? ?parameterdict?
g.sqlnew(property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **`group get`** command. The differences between **`group get`** and **`group sqlnew`** are that the latter forces re-computation of the property data, and that the **SQL** command variant formats the data as **SQL** values rather than for **TCL** or **PYTHON** script processing.

group sqlshow

```
group sqlshow ehandle label propertylist ?filterset? ?parameterdict?
g.sqlshow(property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **`group get`** command. The differences between **`group get`** and **`group sqlshow`** are that the latter does not attempt computation of property data, but raises an error if the data is not present and valid, and that the **SQL** command variant formats the data as **SQL** values rather than for **TCL** or **PYTHON** script processing.

group subcommands

```
group subcommands
dir(Group)
```

Lists all subcommands of the **`group`** command. Note that this command does not require an ensemble handle, or a group label.

group surfaces

```
group surfaces ehandle label ?filterset? ?filtermode?
g.surfaces(?filters=?,?mode=?)
```

Standard cross-referencing command to obtain the labels or references of surface patches the group is associated with. This is explained in more detail in the section about object cross-references.

Example:

```
group surfaces $ehandle $label
```

Note that surface patches do not need to be associated with an atom, and if they are not, they are implicitly not associated with any group.

group xbonds

```
group xbonds ehandle label ?filterset? ?filtermode?
g.xbonds(?filters=?,?mode=?)
```

Retrieve the labels or references of bonds which cross from the group atoms to atoms which are not in the group. This corresponds to the **MDL SDFILE** "`M SBL`" field, except when the group type is a data group, or a type not covered by the **SDF** encoding. This reference command always re-computes these bonds from the group atoms. The original bond set when reading an **SGROUP SDF** is stored in property `G_XBONDS` or `G_BONDS` (for data groups).

In other respects, this is a standard cross-referencing command. The usage of the *filterset* and *filtermode* parameter is explained in the section about object cross-references.

Example:

```
set gh [group create $ehandle [list 1 2 3]]
group xbonds $ehandle $gh
```

gets the bond labels of all crossing bonds.

The hierarchy Command

Hierarchy objects are multi-level ordered trees which store major objects (ensembles, reactions, datasets, tables, networks, biologics) in their leave nodes and contain one or more levels of hierarchy nodes which organize the object content and may themselves possess property data. Hierarchy nodes may contain, in addition to the leaf objects, other hierarchy nodes.

Hierarchy objects are major objects. Associated properties start with a H_ prefix.

hierarchy add

```
hierarchy add hhandle ?handle?...
h.add(?handle/ref?,...)
```

Add objects to a hierarchy. The handles can be ensembles, reactions, datasets, tables, networks, biologics or hierarchies. If an added object is a hierarchy, it is added as a subtree. The new objects are added at the end of the current object set in the hierarchy. It is possible to use this command to change the position of an object within a hierarchy, or to transfer it from a different hierarchy.

In addition to recognized handles, the arguments may also be identifiable reaction or structure line encodings, such as **SMILES** or **REACTION SMILES** strings. These objects are decoded with standard options, get their own handles, and immediately added to the hierarchy.

The command returns the hierarchy handle or reference.

hierarchy append

```
hierarchy append hhandle ?property value?...
h.append({?property:value,?...})
h.append(?property,value,?...)
```

Standard data manipulation command for appending property data. It is explained in more detail in the section about setting property data.

The command returns the first data value.

hierarchy assign

```
hierarchy assign hhandle srcproperty dstproperty
h.assign(srcproperty=,dstproperty=)
```

Assign property data to another property on the same hierarchy. Both properties must be associated with the hierarchy object class. This process is more efficient than going through a pair of **`hierarchy get/hierarchy set`** commands, because in most cases no string or **TCL/PYTHON** script object representations of the property data need to be created.

Both source and destination properties may be addressed with field specifications. A data conversion path must exist between the data types of the involved properties. If any data conversion fails, the command fails. For example, it is possible to assign a string property to a numeric property - but only if all property values can be successfully converted to that numeric type. The reverse example case always succeeds, out-of-memory errors and similar global events excluded.

The original property data remains valid. The command variant **`hierarchy rename`** directly exchanges the property name without any data duplication or conversion, if that is possible. In any case, the original property data is no longer present after the execution of this command variant.

The command returns the object handle for **Tcl**, or object reference for **Python**.

If the object class of the assigned property is not hierarchy, the command executes recursively on all hierarchy objects.

Examples:

```
hierarchy assign $hh H_IDENT H_NAME
hierarchy assign $hh E_IDENT E_NAME
```

hierarchy biologics

```
hierarchy biologics hhandle ?filterlist? ?recursive?
h.biologics(?filters=?,?recursive=?)
```

Return a list of the handles or references of all biologics objects in the hierarchy node which additionally pass the filter set, if one is specified.

By default, only those items directly stored on the command object are listed, but this can be changed via the boolean *recursive* flag. If it is set, the command additionally traverses all hierarchy objects below the current one. Recursively found objects are appended to the list without a level indication. The hierarchy node they are attached to, and from there its level `H_LEVEL` or other property value, can be obtained by the object's **`hierarchy`** command.

Example:

```
hierarchy biologics $hhandle
```

hierarchy create

```
hierarchy create ?handle?...
Hierarchy(?handle/ref?,...)
Hierarchy.Create(?handle/ref?,...)
```

Create a new hierarchy object with an initial set of embedded objects.

The arguments may be handles of structure ensembles, reactions, datasets, networks, tables, or hierarchies themselves.

In addition to recognized handles, the arguments may also be identifiable reaction or structure line encodings, such as **SMILES** or **Reaction SMILES** strings. These objects are decoded with standard options, get their own handles, and immediately added to the hierarchy.

The command returns the handle or reference of the newly created hierarchy object.

hierarchy dataset

```
hierarchy dataset hhandle ?filterlist?
h.dataset(?filters=?)
```

Return the dataset handle or reference of the dataset the hierarchy is a member of. It the hierarchy is not member of a dataset, or does not pass all of the optional filters, an empty string or **`None`** for **Python** is returned.

This command is not the same as **`hierarchy datasets`**.

Example:

```
hierarchy dataset $hhandle
```

hierarchy datasets

```
hierarchy datasets hhandle ?filterlist? ?recursive?
h.biologics(?filters=?,?recursive=?)
```

Return a list of the handles or references of all dataset objects in the hierarchy node which additionally pass the filter set, if one is specified.

By default, only those items directly stored on the command object are listed, but this can be changed via the boolean *recursive* flag. If it is set, the command additionally traverses all hierarchy objects below the current one. Recursively found objects are appended to the list without a level indication. The hierarchy node they are attached to, and from there its level `H_LEVEL` or other property value, can be obtained by the object's **`hierarchy`** command.

This is not the same as the **`hierarchy dataset`** command.

Example:

```
hierarchy datasets $hhandle
```

hierarchy defined

```
hierarchy defined hhandle property
h.defined(property)
```

This command checks whether a property is defined for the hierarchy. This is explained in more detail in the section about property validity checking. Note that this is *not* a check for the presence of property data! The **`hierarchy valid`** command is used for this purpose.

hierarchy delete

```
hierarchy delete ?hhandle/hhandlelist/all?...
h.delete()
Hierarchy.Delete("all")
Hierarchy.Delete(?href/hrefsequence/hhandle?,...)
```

Delete hierarchy objects. The special parameter *all* may be used to delete all hierarchies currently registered in the application. Alternatively, any number of hierarchy handles may be specified for specific object deletions.

The command returns the number of deleted hierarchies.

Example:

```
hierarchy delete all
hierarchy delete $hhandle
```

hierarchy dget

```
hierarchy dget hhandle propertylist ?filterset? ?parameterdict?
h.dget(property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **`network get`** command. The difference between **`hierarchy get`** and **`hierarchy dget`** is that the latter does not attempt computation of property data, but rather initializes the property values to the default and return that default if the data is not yet available. For data already present, **`hierarchy get`** and **`hierarchy dget`** are equivalent.

hierarchy dup

```
hierarchy dup hhandle ?dataset? ?position?
h.dup(?target=?,?position=?)
```

Duplicate a hierarchy with all objects in them and all attached data on the hierarchy object proper and its contained objects.

The duplicate hierarchy is placed into the same dataset as the source, if it is a member of a dataset. Specifying an explicitly empty dataset argument (or **None** for **PYTHON**) places the duplicate outside any dataset, regardless of the dataset membership of the source hierarchy.

If the duplicate is moved to a dataset, it is appended to the dataset end by default. This happens also if the position parameter is explicitly specified as *end* or an empty string. Otherwise, the hierarchy is inserted at the given position, starting with 0. If the requested position is larger than the current size of the dataset, the hierarchy is appended.

Example:

```
hierarchy dup $hhandle
```

The command returns a new hierarchy handle or reference.

hierarchy ens

```
hierarchy ens hhandle ?filterlist? ?recursive?
h.ens(?filters=?,?recursive=?)
```

Return a list of the handles or references of all structure ensemble objects in the hierarchy node which additionally pass the filter set, if one is specified.

By default, only those items directly stored on the command object are listed, but this can be changed via the boolean *recursive* flag. If it is set, the command additionally traverses all hierarchy objects below the current one. Recursively found objects are appended to the list without a level indication. The hierarchy node they are attached to, and from there its level `H_LEVEL` or other property value, can be obtained by the object's **hierarchy** command.

Example:

```
hierarchy ens $hhandle
```

hierarchy exists

```
hierarchy exists hhandle ?filterlist?
h.exists(?filters=?)
Hierarchy.Exists(href,?filters=?)
```

Check whether a hierarchy handle or reference is valid. The command returns boolean 0 or 1. Optionally, the hierarchy may be filtered by a standard filter list, and if it does not pass the filter, it is reported as not valid.

Example:

```
hierarchy exists $hhandle
```

hierarchy expr

```
hierarchy expr hhandle expression
h.expr(expression)
```

Compute a standard **SQL**-style property expression for the hierarchy. This is explained in detail in the chapter on property expressions.

hierarchy fill

```
hierarchy fill hhandle ?property value?...
h.fill({?property:value,...})
h.fill(?property,value?,...)
```

Standard data manipulation command for setting data, ignoring possible mismatches between the lengths of the lists of objects associated with the property and the value list. It is explained in more detail in the section about setting property data.

hierarchy filter

```
hierarchy filter hhandle filterlist
h.filter(filters=)
```

Check whether the hierarchy passes a filter list. The return value is boolean 1 for success and 0 for failure.

hierarchy get

```
hierarchy get hhandle propertylist ?filterset? ?parameterdict?
hierarchy get hhandle attribute
h.get(property=,?filters=?,?parameters=?)
h.get(attribute)
h[property/attribute]
h.property/attribute
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

Examples:

```
hierarchy get $hhandle {H_IDENT H_NAME}
```

yields the ID and name of the hierarchy as a list. If the information is not available, an attempt is made to compute it. If the computation fails, an error results.

For the use of the optional property parameter list argument, refer to the documentation of the **ens get** command.

Variants of the **hierarchy get** command are **hierarchy new, hierarchy dget, hierarchy jget, hierarchy jnew, hierarchy jshow, hierarchy nget, hierarchy show, hierarchy sqldget, hierarchy sqlget, hierarchy sqlnew,** and **hierarchy sqlshow.**

In addition to property data, a hierarchy object possesses a few attributes, which can be retrieved with the *get* command (but not by its related sister subcommands like **dget**, **sqlget**, etc.). Some of them are also modifiable via **hierarchy set.** These attributes are:

- *coords*
  If the toolkit was compiled with factory support, these are the coordinates of the object icon on its workbench, encoded as integer pair. This attribute can be changed.

- *deletable*
  Flag indicating whether the object can be deleted with a standard **`hierarchy delete`** command. This attribute is read-only. Objects which are, for example, property data values cannot be deleted by standard means.
- *failures*
  If the property computation failure cache is active, return a list of all properties which have failed computation for this object after the last structural change. This attribute is read-only.
- *footer*
  If the toolkit was compiled with factory support, this is the footer of the object icon on a workbench. This attribute can be changed.
- *gflags*
  If the toolkit was compiled with factory support, this is the currently set object icon rendering flag collection.
- *header*
  If the toolkit was compiled with factory support, this is the header of the object icon on a workbench. This attribute can be changed.
- *hidden*
  Flag indicating whether the object is hidden. This is not the same as the *invisible* state. This attribute is intended to be used for rendering selections. This attribute can be changed.
- *incomplete*
  Boolean status flag indicating an aborted input operation during the read of the object from file, which returned the structure intact but without the complete set of associated data. An aborted input may be either be the result of an explicitly set input control flag, or by encountering property data which could not be decoded. This attribute is read-only.
- *invisible*
  Flag indicating whether the object is invisible. This is not the same as the *hidden* state. An invisible object is no longer accessible via its handle. This is usually the case for objects which are scheduled for deletion, but still have lingering pointer references. This attribute is read-only.
- *javaobject*
  If the toolkit was compiled with **JNI** support, this attribute reports the memory address of the **JNI** wrapper class instance, if it exists.
- *modcount*
  Object data modification count. This attribute is read-only.
- *mutexcount*
  The number of recursive mutex locks held for this object. Only supported on Linux.
- *pyobject*
  If the toolkit was compiled with Python support, this attribute reports the memory address of the Python wrapper class instance, if it exists. This attribute is read-only.
- *pyrefcount*
  If the toolkit was compiled with Python support, this attribute reports the reference count of the Python wrapper class instance, if it exists. This attribute is read-only.

- *refcount*
  If the **Tcl** interpreter is using native **Cactvs** objects instead of string-based major object handles and integer-based minor object labels to identify toolkit objects, this returns the number of **Tcl** object references active for this object. This attribute is read-only.
- *scoped*
  A boolean object visibility control flag. If set, and global control flag `::cactvs(object_scope)` is also set, the object is visible only in the**Tcl** interpreter which set the scope flag and thus claimed it. Object list commands executed in other interpreters omit this object, and attempts to decode its handle in other interpreters will fail. The most common use of this feature is the hiding of persistent chemistry objects in scripted property computation functions.
- *selected*
  Flag indicating whether the object is selected. This attribute can be changed.
- *tooltip*
  If the toolkit was compiled with factory support, this is the tooltip of the object icon on a workbench. This attribute can be changed.
- *uuid*
  An automatically generated **UUID** globally identifying the object. This attribute is read-only, different for every object, and not dependent on its contents.
- *x*
  If the toolkit was compiled with factory support, this is the *x* coordinate of the object icon on its workbench. This attribute can be changed.
- *y*
  If the toolkit was compiled with factory support, this is the *y* coordinate of the object icon on its workbench.This attribute can be changed.

hierarchy getparam

```
hierarchy getparam hhandle property ?key? ?default?
h.getparam(property=,?key=?,?default=?)
```

Retrieve a named computation parameter from valid property data. If the key is not present in the parameter list, an empty string is returned (**None** for **Python**). If the default argument is supplied, that value is returned in case the key is not found.

If the key parameter is omitted, a complete set of the parameters used for computation of the property value is returned in dictionary format.

This command does not attempt to compute property data. If the specified property is not present, an error results.

hierarchy hdup

```
hierarchy hdup hhandle ?dataset? ?position?
h.hdup(?target=?,?position=?)
```

Duplicate a hierarchy with all objects in them and all attached data on the hierarchy object proper and its contained objects. Additionally, hydrogen addition is performed on all objects which support this operation.

The duplicate hierarchy is placed into the same dataset as the source, if it is a member of a dataset. Specifying an explicitly empty dataset argument (or **`None`** for **PYTHON**) places the duplicate outside any dataset, regardless of the dataset membership of the source hierarchy.

If the duplicate is moved to a dataset, it is appended to the dataset end by default. This happens also if the position parameter is explicitly specified as *end* or an empty string. Otherwise, the hierarchy is inserted at the given position, starting with 0. If the requested position is larger than the current size of the dataset, the hierarchy is appended.

Example:

```
hierarchy hdup $hhandle
```

The command returns a new hierarchy handle or reference.

hierarchy hierarchies

```
hierarchy hierarchies hhandle ?filterlist? ?recursive?
h.hierarchies(?filters=?,?recursive=?)
```

Return a list of the handles or references of all hierarchy objects in the hierarchy node which additionally pass the filter set, if one is specified.

By default, only those items directly stored on the command object are listed, but this can be changed via the boolean *recursive* flag. If it is set, the command additionally traverses all hierarchy objects below the current one. Recursively found objects are appended to the list without a level indication. The hierarchy node they are attached to, and from there its level `H_LEVEL` or other property value, can be obtained by the object's **`hierarchy`** command.

This command is not the same as the **`hierarchy hierarchy`** command.

Example:

```
hierarchy hierachies $hhandle
```

hierarchy hierarchy

```
hierarchy hierarchy hhandle ?filterlist? ?root?
h.hierarchy(?filters=?,?root=?)
```

Return the hierarchy handle or reference of the hierarchy the command object is part of. If the command object is the root node, or does not pass all of the optional filters, an empty string or **`None`** for **PYTHON** is returned. By default, the hierarchy object which directly contains the command object is returned. If the *root* flag is set, the root hierarchy object is reported instead, which is the same only if the hierarchy has only a single level.

This is not the same as the **`hierarchy hierarchies`** command.

Example:

```
hierarchy hierarchy $hhandle
```

hierarchy index

```
hierarchy index hhandle
h.index()
```

Get the position of the hierarchy in the object list of its dataset. If the object is not member of a dataset, -1 is returned.

hierarchy jget

```
hierarchy jget hhandle propertylist ?filterset? ?parameterdict?
h.jget(property=,?filters=?,?parameters=?)
```

This is a variant of **`hierarchy get`** which returns the result data as a **JSON** formatted string instead of **Tcl** or **Python** interpreter objects. The command is usable only for property data, not attribute retrieval.

hierarchy jnew

```
hierarchy jnew hhandle propertylist ?filterset? ?parameterdict?
h.jnew(property=,?filters=?,?parameters=?)
```

This is a variant of **`hierarchy new`** which returns the result data as a **JSON** formatted string instead of **Tcl** or **Python** interpreter objects.

hierarchy jshow

```
hierarchy jshow hhandle propertylist ?filterset? ?parameterdict?
h.jshow(property=,?filters=?,?parameters=?)
```

This is a variant of **`hierarchy show`** which returns the result data as a **JSON** formatted string instead of **Tcl** or **Python** interpreter objects.

hierarchy list

```
hierarchy list ?filterlist?
Hierarchy.List(?filters=?)
```

This command returns a list of the hierarchy handles currently registered in the application. This list may optionally be filtered by a standard filter list.

hierarchy lock

```
hierarchy lock hhandle propertylist/hierarchy/all ?compute?
h.lock(property=,?compute=?)
```

Lock property data of the hierarchy object, meaning that it is no longer subject to the standard data consistency manager control. The data consistency manager deletes specific property data if anything is done to the hierarchy which would invalidate the information. Property data remains locked until is it explicitly unlocked.

The property data to lock can be selected by providing a list of the following identifiers:

- Property names
  Valid property instances on the hierarchy object are locked. If the boolean *compute* flag is set, an attempt is made to compute the property if it is not yet present. Otherwise, a request to lock non-existent data is silently ignored. It is not possible to lock individual property fields.
- *all*
  All valid hierarchy properties are locked. The compute flag is ignored.

- *hierarchy*
  This is an object class identifier. All property data which is controlled by the hierarchy major object and attached to the specified object class is locked. Since hierarchies do not contain minor objects, this identifier is equivalent to *all*.

The lock can be released by a **`hierarchy unlock`** command.

The return value is the original hierarchy handle or reference.

hierarchy max

```
hierarchy max hhandle propertylist ?filterset?
h.max(property=,?filters=?)
```

Get the maximum value of one or more properties in from the elements in the hierarchy. The property argument may be any property attached to hierarchy members, or minor objects thereof. If the *filterset* argument is specified, the maximum value is searched only for objects which pass the filter set.

hierarchy metadata

```
hierarchy metadata hhandle property ?field ?value??
h.metadata(property=,?field=?,?value=?)
```

Obtain property metadata information, or set it. The handling of property metadata is explained in more detail in its own introductory section. The related commands **`hierarchy setparam`** and **`hierarchy getparam`** can be used for convenient manipulation of specific keys in the computation parameter field. Metadata can only be read from or set on valid property data.

hierarchy min

```
hierarchy min hhandle propertylist ?filterset?
h.min(property=,?filters=?)
```

Get the minimum value of one or more properties in from the elements in the hierarchy. The property argument may be any property attached to hierarchy members, or minor objects thereof. If the *filterset* argument is specified, the maximum value is searched only for objects which pass the filter set.

hierarchy move

```
hierarchy move hhandle ?datasethandle|remotehandle? ?position?
h.move(?target=?,?position=?)
```

Make the hierarchy a member of a dataset, or remove it from a dataset. If the dataset handle or reference parameter is omitted, or is an empty string, or **`None`** for **PYTHON**, the object is removed from its current dataset. The dataset handle or reference may be the name of a remote dataset for moving objects over a network connection.

If a target dataset handle or reference is specified, the object is added to the dataset, if allowed by the acceptance bits of the dataset, and removed from any dataset it was member of before the execution of the command. By default the object is added to the end of the dataset object list, but the final optional parameter allows the specification of a dataset object list index. The first position is index zero. If the parameter value *end* is used, or the index is bigger than the current number of dataset objects minus one, the hierarchy is appended as per the default. It is legal to use this command for moving objects within the same dataset.

Another special position value is *random* or *rnd*. This value moves to the object to a random position in the dataset. Using this mode with remote datasets is currently not supported.

By default, datasets do not accept hierarchies as objects. This must be explicitly enabled by modifying the acceptance bits, as for example in

```
dataset append $dhandle accepts hierarchy
```

The dataset handle cannot be a transient dataset.

The return value of the command is the dataset of the object prior to the move operation. It is either a dataset handle/reference, or an empty string (**Tcl**) or **None** (**Python**) if it was not member of a dataset.

hierarchy mutex

```
hierarchy mutex hhandle mode
h.mutex(mode)
```

Manipulate the object mutex. During the execution of a script command, the mutex of the major object(s) associated with the command are automatically locked and unlocked, so that the operation of the command is thread-safe. This applies to builds that support multi-threading, either by allowing multiple parallel script interpreters in separate threads or by supporting helper threads for the acceleration of command execution or background information processing.

This command locks major objects for a period of time that exceeds a single command. A lock on the object can only be released from the same interpreter thread that set the lock. Any other threaded interpreters, or auxiliary threads, block until a mutex release command has been executed when accessing a locked command object. This command supports the following modes:

- *lock*
  Increase the recursive mutex lock count on the object. The command returns the current lock count after the command, excluding the transient single-command lock.
- *reset*
  Release all persistent locks on the object, if they exist.
- *test*
  Return the current persistent lock count on the object. This excludes the transient per-command lock.
- *unlock*
  Decrease the recursive lock count on the object. The command returns the current lock count after the command, excluding the transient single-command lock. Unlocking an object which has not been persistently locked results in an error.

There is no *trylock* command variant because the command already needs to be able to acquire a transient object mutex lock for its execution.

The command returns the current lock count.

hierarchy need

```
hierarchy need hhandle propertylist ?mode? ?parameterdict?
h.need(property=,?mode=?,?parameters=?)
```

Standard command for the computation of property data, without immediate retrieval of results. This command is explained in more detail in the section about retrieving property data.

The return value is the original hierarchy handle or reference.

hierarchy networks

```
hierarchy networks hhandle ?filterlist? ?recursive?
h.networks(?filters=?,?recursive=?)
```

Return a list of the handles or references of all network objects in the hierarchy node which additionally pass the filter set, if one is specified.

By default, only those items directly stored on the command object are listed, but this can be changed via the boolean *recursive* flag. If it is set, the command additionally traverses all hierarchy objects below the current one. Recursively found objects are appended to the list without a level indication. The hierarchy node they are attached to, and from there its level `H_LEVEL` or other property value, can be obtained by the object's **`hierarchy`** command.

Example:

```
hierarchy networks $hhandle
```

hierarchy new

```
hierarchy new hhandle propertylist ?filterset? ?parameterdict?
h.new(property=,?mode=?,?parameters=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **`hierarchy get`** command. The difference between **`hierarchy get`** and **`hierarchy new`** is that the latter forces the re-computation of the property data, regardless whether it is present and valid, or not.

hierarchy nget

```
hierarchy nget hhandle propertylist ?filterset? ?parameterdict?
h.nget(property=,?mode=?,?parameters=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **`hierarchy get`** command. The difference between **`hierarchy get`** and **`hierarchy nget`** is that the latter always returns numeric data, even if symbolic names for the values are available.

hierarchy nnew

```
hierarchy nnew hhandle propertylist ?filterset? ?parameterdict?
h.nnew(property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading object data and attributes. It is explained in more detail in the section about retrieving property data.

For examples, see the **`hierarchy get`** command. The difference between **`hierarchy get`** and **`hierarchy nnew`** is that the latter always returns numeric data, even if symbolic names for the values are available, and that property data re-computation is enforced.

hierarchy objects

```
hierarchy objects hhandle ?filterlist? ?recursive?
h.objects(?filters=?,?recursive=?)
```

Return a list of the handles or references of all objects in the hierarchy node which pass the filter set, if one is specified. Lower hierarchy node objects are not listed.

By default, only those items directly stored on the command object are listed, but this can be changed via the boolean *recursive* flag. If it is set, the command additionally traverses all hierarchy objects below the current one. Recursively found objects are appended to the list without a level indication. The hierarchy node they are attached to, and from there its level `H_LEVEL` or other property value, can be obtained by the object's **`hierarchy`** command.

Example:

```
hierarchy objects $hhandle
```

hierarchy pack

```
hierarchy pack hhandle ?maxsize? ?requestprops? ?suppressedprops?
?compressionlib?
h.pack(?maxsize=?,?requestprops=?,?suppressedprops=?,?compressionlib=?)
```

Pack the hierarchy object into a base64-encoded compressed serialized object string. This string does not contain any non-printable characters and is a full dump of the internal state of the object, omitting only property data that was declared to be so easily re-computed that a dump is not worthwhile. The objects in the hierarchy and their property data are part of the dump. Further object relationships, such as datasets the object might be a member in are not saved.

The maximum size of the object string (default -1, meaning unlimited size) can be configured by the optional *maxsize* parameter. The size is specified in bytes. If the pack string would be longer than the maximum size, an error results.

The other optional property parameter lists allow to request a specific property set to be part of the package, even if it normally would not be included, and to explicitly omit properties from the dump. No property computation is performed, and suppressed properties are not purged from the hierarchy.

Hierarchies can be restored from a packed object string by the **`hierarchy unpack`** command.

The hierarchy object and its contained objects remain in existence after using this command.

The default compression library is *zlib*. Other useful variants include *lzo* and *gzip* (and there are other internal types), but these may not be available on all builds due to license issues, and you need to specify the compression library when a dataset is unpacked. It is generally recommended to stay with *zlib*.

The return value of this command is the packed string.

In **Python**, hierarchy objects support the standard *pickle/unpickle* protocol.

Example:

```
set dbstring [hierarchy pack $hhandle]
```

hierarchy properties

```
hierarchy properties hhandle ?pattern? ?noempty?
h.properties(?pattern=?,?noempty=?)
```

Generate a list of the names of all properties attached to the hierarchy object. Optionally, the list may be filtered by a string match pattern.

If the *noempty* flag is set, only properties where at least one data element is not the property default value are output. By default, the filter pattern is an empty string, and the *noempty* flag is not set.

The command may be abbreviated to **`props`** instead of the full name **`properties`**.

hierarchy purge

```
hierarchy purge hhandle propertylist/hierarchy/specialname ?emptyonly?
h.purge(properties=,?emptyonly?)
```

Delete property data from the hierarchy. If a property marked for deletion is not present on an object, it is silently ignored. If the hierarchy is not a dataset member, a request for the deletion of dataset properties is also ignored.

If the object class name *hierarchy* is used instead of a specific property name, all hierarchy property data (`H_` prefix) is deleted from the object.

The optional boolean flag *emptyonly* can be used to restrict the deletion to those properties where all the values for a property associated with a major object (such as on all atoms in an ensemble for atom properties, or just the single ensemble property value for ensemble properties) are set to the default property value.

The return value is the original object handle or reference.

hierarchy reactions

```
hierarchy reactions hhandle ?filterlist? ?recursive?
h.reactions(?filters=?,?recursive=?)
```

Return a list of the handles or references of all reaction objects in the hierarchy node which additionally pass the filter set, if one is specified.

By default, only those items directly stored on the command object are listed, but this can be changed via the boolean *recursive* flag. If it is set, the command additionally traverses all hierarchy objects below the current one. Recursively found objects are appended to the list without a level indication. The hierarchy node they are attached to, and from there its level `H_LEVEL` or other property value, can be obtained by the object's **`hierarchy`** command.

Example:

```
hierarchy reactions $hhandle
```

hierarchy ref

```
Hierarchy.Ref(identifier)
```

Python only method to get a hierarchy reference from a handle or another identifier. For hierarchies, other recognized identifiers are hierarchy references, integers encoding the numeric part of the handle string, or the **UUID** of the hierarchy object.

hierarchy remove

```
hierarchy remove hhandle ?objhandle?...
h.remove(?oref/ohandle?,...)
```

Remove objects from the hierarchy. The object arguments must be a member of the command hierarchy object.

The command returns the original handle or reference.

hierarchy rename

```
hierarchy rename hhandle srcproperty dstproperty
h.rename(srcproperty=,dstproperty=)
```

This is a variant of the **hierarchy assign** command. Please refer the command description in that paragraph.

hierarchy set

```
hierarchy set hhandle ?property value?...
h.set(property,value,...)
h.set({property:value,...})
h.property = value
h[property] = value
```

Standard data manipulation command. It is explained in more detail in the section about setting property data.

hierarchy setparam

```
hierarchy setparam hhandle property ?key value?...
hierarchy setparam hhandle property dictionary
h.setparam(property,?key,value?...)
h.setparam(property,dict)
```

Set or update a property computation parameter in the metadata parameter list of a valid property. This command is described in the section about retrieving property data. The current settings of the computation parameters in the property definition are not changed.

The return value is the updated property computation parameter dictionary.

hierarchy show

```
hierarchy show hhandle propertylist ?filterset? ?parameterdict?
h.show(property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **`hierarchy get`** command. The difference between **`hierarchy get`** and **`hierarchy show`** is that the latter does not attempt computation of property data, but raises an error if the data is not present and valid. For data already present, **`hierarchy get`** and **`hierarchy show`** are equivalent.

hierarchy sqldget

```
hierarchy sqldget hhandle propertylist ?filterset? ?parameterdict?
h.sqldget(property=,?mode=?,?parameters=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **`hierarchy get`** command. The differences between **`hierarchy get`** and **`hiedrarchy sqldget`** are that the latter does not attempt computation of property data, but initializes the property value to the default and returns that default, if the data is not present and valid; and that the **SQL** command variant formats the data as **SQL** values rather than for **TCL** or **PYTHON** script processing.

hierarchy sqlget

```
hierarchy sqlget hhandle propertylist ?filterset? ?parameterdict?
h.sqldget(property=,?mode=?,?parameters=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **`hierarchy get`** command. The difference between **`hierarchy get`** and **`hierarchy sqlget`** is that the **SQL** command variant formats the data as **SQL** values rather than for **TCL** or **PYTHON** script processing.

hierarchy sqlnew

```
hierarchy sqlnew hhandle propertylist ?filterset? ?parameterdict?
h.sqlnew(property=,?mode=?,?parameters=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **`hierarchy get`** command. The differences between **`hierarchy get`** and **`hierarchy sqlnew`** are that the latter forces re-computation of the property data, and that the **SQL** command variant formats the data as **SQL** values rather than for **TCL** or **PYTHON** script processing.

hierarchy sqlshow

```
hierarchy sqlshow hhandle propertylist ?filterset? ?parameterdict?
h.sqlshow(property=,?mode=?,?parameters=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **`hierarchy get`** command. The differences between **`hierarchy get`** and **`hierarchy sqlshow`** are that the latter does not attempt computation of property data, but raises an error if the data is not present and valid, and that the **SQL** command variant formats the data as **SQL** values rather than for **TCL** or **PYTHON** script processing.

hierarchy subcommands

```
hierarchy subcommands
dir(Hierarchy)
```

Lists all subcommands of the **hierarchy** command. Note that this command does not require a hierarchy handle.

hierarchy tables

```
hierarchy tables hhandle ?filterlist? ?recursive?
h.tables(?filters=?,?recursive=?)
```

Return a list of the handles or references of all table objects in the hierarchy node which additionally pass the filter set, if one is specified.

By default, only those items directly stored on the command object are listed, but this can be changed via the boolean *recursive* flag. If it is set, the command additionally traverses all hierarchy objects below the current one. Recursively found objects are appended to the list without a level indication. The hierarchy node they are attached to, and from there its level `H_LEVEL` or other property value, can be obtained by the object's **hierarchy** command.

Example:

```
hierarchy tables $hhandle
```

hierarchy transfer

```
hierarchy transfer hhandle propertylist ?targethandle? ?targetpropertylist?
h.transfer(properties=,?target=?,?targetproperties=?)
```

Copy property data from one hierarchy to another hierarchy or other major object, without going through an intermediate scripting language object representation, or dissociate property data from the hierarchy. If a property in the argument property list is not already valid on the source reaction, an attempt is made to compute it.

If a target property list is given, the data from the source is stored as content of a different property on the target. For this, the data types of the properties must be compatible, and the object class of the target property that of the target object. No attempt is made to convert data of mismatched types. In case of multiple properties, the source property list and the target property list are stepped through in parallel. If there is no target property list, or it is shorter than the source list, unmatched entries are stored as original property values, and this implies that the object class of the source and target objects are the same.

If no target object is specified, or it is spelled as an empty string or **Python None**, the visible effect of the command is the same as a simple **hierarchy get**, i.e. the result is the property data value or value list. The property data is then deleted from the source object. In case the data type of the deleted property was a major object (i.e. an ensemble, reaction, table, dataset or network), it is only unlinked from the source object, but not destroyed. This means that the object handles returned by the command can henceforth the used as independent objects. They can be deleted by a normal object deletion command, and are no longer managed by the source object.

hierarchy unlock

```
hierarchy unlock hhandle propertylist/hierarchy/all
h.unlock(property=)
```

Unlock property data for the reaction, meaning that they are again under the control of the standard data consistency manager.

The property data to unlock can be selected by providing a list of the following identifiers:

- Property names or references
  Valid property instances on the hierarchy are unlocked. Non-existent data is silently ignored. It is not possible to unlock individual property fields.
- *all*
  All valid hierarchy properties are unlocked.
- *hierarchy*
  This is an object class identifier. All property data which is controlled by the hierarchy major object and attached to the specified object class is unlocked. Since hierarchies do not contain minor objects, this identifier is equivalent to *all*.

Property data locks are obtained by the **`hierarchy lock`** command.

The return value is the original hierarchy handle or reference.

hierarchy unpack

```
hierarchy unpack packstring ?compressionlib?
Hierarchy.Unpack(data=,?compressionlib=?)
```

Unpack a base64-encoded serialized object string which was created by a **`hierarchy pack`** command. The return value of this function is the handle or reference of the newly created hierarchy object, which is an exact duplicate of the packed original hierarchy.

The default compression library is *zlib*. For more options, see **`reaction pack`**.

Example:

```
set packdata [hierarchy pack [hierarchy create C=O>>CO]]
set hhandle [hierarchy unpack $packdata]
```

hierarchy valid

```
hierarchy valid hhandle propertylist
h.valid(property/propertysequence)
```

Returns a list of boolean values indicating whether values for the named properties are currently set for the hierarchy. No attempt at computation is made. For **PYTHON**, where single-item lists are syntactically not the same as a single value, the return value is a single boolean if the argument was a string or a property reference, and only a single property was decoded.

`hierarchy has` is an alias to this command.

hierarchy verify

```
hierarchy verify hhandle property
h.verify(property)
```

Verify the values of the specified property on the hierarchy. The property data must be valid, and a hierarchy property. If the data can be found, it is checked against all constraints defined for the property, and, if such a function has been defined, is tested with the value verification function of the property.

If all tests are passed, the return value is boolean 1, 0 if the data could be found but fails the tests, and an error condition otherwise.

The majorobj Command

This command provides simple, generic versions of some commonly used major objects (ensembles, reactions, datasets, molfiles, tables, networks) commands. The difference to the object-specific commands is that any major object handle is accepted as object identifier, not just the type of handle associated with the associated specialized object command. Specialized object commands generally implement more powerful commands with additional, usually class-specific options. This command is generally used when different types of objects (such as ensembles and reactions) share a common simple processing path and code duplication for each type of object would be tedious.

It is not possible to create generic major objects without specialization - this is an abstraction.

majorobj delete

```
majorobj delete ?handle?...
Majorobj.Delete(?mrefsequence/mref/handle?,...)
```

Delete major objects.

The command returns the number of deleted objects.

Example:

```
majorobj delete $ehandle $xhandle
```

majorobj dget

```
majorobj dget handle propertylist ?filterset? ?parameterdict?
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **`majorobj get`** command. The difference between **`majorobj get`** and **`majorobj dget`** is that the latter does not attempt computation of property data, but rather initializes the property values to the default and return that default if the data is not yet available. For data already present, **`majorobj get`** and **`majorobj dget`** are equivalent.

A **Python** implementation would not be useful because every major object has a **`dget()`** method, which can be invoked without exact knowledge of the object class.

majorobj dup

```
majorobj dup handle ?dataset? ?position?
```

Duplicate a major object.

The duplicate object is placed into the same dataset as the source, if it is a member of a dataset. Specifying an explicitly empty dataset argument places the duplicate outside any dataset, regardless of the dataset membership of the source object.

If the duplicate is moved to a dataset, it is appended to the dataset end by default. This happens also if the position parameter is explicitly specified as *end* or an empty string. Otherwise, the object is inserted at the given position, starting with 0. If the requested position is larger than the current size of the dataset, the object is appended.

A **PYTHON** implementation would not be useful because every major object has a **`dup()`** method, which can be invoked without exact knowledge of the object class.

Example:

```
majorobj dup $handle
```

The command returns a new object handle or reference.

majorobj exists

```
majorobj exists handle ?filterlist?
```

Check whether a handle is valid. The command returns boolean 0 or 1. Optionally, the object may be filtered by a standard filter list, and if it does not pass the filter, it is reported as not valid.

A **PYTHON** implementation would not be useful because every major object has a **`exists()`** method, which can be invoked without exact knowledge of the object class.

Example:

```
majorobj exists $handle
```

majorobj get

```
majorobj get handle propertylist ?filterset? ?parameterdict?
majorobj get handle attribute
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For the use of the optional property parameter list argument, refer to the documentation of the **`ens get`** command.

Variants of the **`majorobj get`** command are **`majorobj new, majorobj dget, majorobj jget, majorobj jnew, majorobj jshow, majorobj nget, majorobj show, majorobj sqldget, majorobj sqlget, majorobj sqlnew,`** and **`majorobj sqlshow`**.

In addition to property data, all major objects possess a few common attributes, which can be retrieved with the **`get`** command (but not by its related sister subcommands like **`dget, sqlget,`** etc.). Object-specific attributes can only be retrieved via the object-specific access commands. The common attributes are:

- *coords*
  If the toolkit was compiled with factory support, these are the coordinates of the object on its workbench, encoded as integer pair.
- *deletable*
  Flag indicating whether the object can be deleted with a standard **`majorobj delete`** command. This attribute is read-only. Object which are, for example, property data values or a part of a **`molfile loop`** command cannot be deleted by standard means.
- *failures*
  A list of properties for which computation failed on this object. This is a read-only attribute. Depending on configuration settings, this information may be used to block pointless attempts at re-computation of incomputable data.

- *footer*
  If the toolkit was compiled with factory support, this is the footer of the object icon on a workbench.
- *gflags*
  If the toolkit was compiled with factory support, this is the currently set object icon rendering flag collection.
- *handle*
  The handle of the object, always reported as a string.
- *header*
  If the toolkit was compiled with factory support, this is the header of the object icon on a workbench.
- *hidden*
  Flag indicating whether the object is hidden. This is not the same as the *invisible* state. This attribute is intended to be used for rendering selections.
- *invisible*
  Flag indicating whether the object is invisible. This is not the same as the *hidden* state. An invisible object is no longer accessible via its handle. This is usually the case for objects which are scheduled for deletion, but still have lingering referring pointers.
- *javaobject*
  If the toolkit was compiled with **JNI** support, this attribute reports the memory address of the **JNI** wrapper class instance, if it exists.
- *modcount*
  Object modification count.
- *mutexcount*
  The number of recursive mutex locks held for this object. Only supported on Linux.
- *pyobject*
  If the toolkit was compiled with Python support, this attribute reports the memory address of the Python wrapper class instance, if it exists.
- *pyrefcount*
  If the toolkit was compiled with Python support, this attribute reports the reference count of the Python wrapper class instance, if it exists.
- *refcount*
  If the **Tcl** interpreter is using native **Cactvs** objects instead of string-based major object handles and integer-based minor object labels to identify toolkit objects, this returns the number of **Tcl** object references active for this object.
- *reference*
  The handle of the object, reported as **Tcl** object reference object if the **Tcl** interpreter is configured to use these.
- *scoped*
  A boolean object visibility control flag. If set, and global control flag `::cactvs(object_scope)` is also set, the network object is visible only in the**Tcl** interpreter which set the scope flag and thus claimed it. Object list commands executed in other interpreters omit this object, and attempts to decode its handle in other interpreters will fail. The most common use of this feature is the hiding of persistent chemistry objects in scripted property computation functions.

- *selected*
  Flag indicating whether the object is selected.
- *tooltip*
  If the toolkit was compiled with factory support, this is the tooltip of the object icon on a workbench.
- *uuid*
  An automatically generated **UUID** globally identifying the object. This attribute is read-only, different for every object, and not dependent on its contents.
- *x*
  If the toolkit was compiled with factory support, this is the x coordinate of the object on its workbench.
- *y*
  f the toolkit was compiled with factory support, this is the y coordinate of the object on its workbench.

A **PYTHON** implementation would not be useful because every major object has a **`get()`** method, which can be invoked without exact knowledge of the object class.

majorobj hadd

```
majorobj hadd handle
```

Add a standard set of hydrogens to the object, if applicable (ensembles, reaction, datasets). The command returns the total number of hydrogens added. This command version has less options than the class-specific variants.

A **PYTHON** implementation would not be useful because all applicable major objects have a **`hadd()`** method, which can be invoked without exact knowledge of the object class.

majorobj hdup

```
majorobj hdup handle ?dataset? ?position?
```

This command is the same as **`majorobj dup`**, except that a full set of hydrogens is added to the duplicated objects if applicable (ensembles, reactions, datasets).

A **PYTHON** implementation would not be useful because all applicable major objects have a **`hdup()`** method, which can be invoked without exact knowledge of the object class.

majorobj hstrip

```
majorobj hstrip handle ?flags?
```

This command removes hydrogens from all applicable objects (ensembles, reactions, datasets). By default, all hydrogen atoms on the object are removed.

The *flags* parameter can be used to make the operation more selective. It may be a list of the following flags:

- *deprotonate*
  If this flag is set, a single proton is removed from the first suitable atom. This command variant triggers a standard atom and bond change property invalidation event, and it always ends processing after removing the first proton. Proton removal decreases the charge of the atom by one. In the reaction command variant, this flag is rarely useful - it is supported for compatibility with the **`atom hstrip`** command
- *keepalphawedge*
  Keep hydrogen atoms which are bonded to an atom which is at the tip of a wedgebond. This flag excludes the case where the bond to the hydrogen atom is the wedge bond - use the *keepwedge* flag to cover this case.
- *keepisotopes*
  Keep hydrogen atoms which are isotope labels (including enriched/depleted ^{1}H).
- *keeporiginal*
  Hydrogen atoms which were not automatically added via a *hadd* command are retained. Note that hydrogen addition commands can be run in a mode which does not leave information about automatic addition - hydrogens added this way will also survive.
- *keepprotons*
  Keep any molecules which consist only of hydrogen atoms (such as protons, hydride anions, and molecular hydrogen).
- *keepspecial*
  If this flag is set, hydrogens which are usually displayed, such as on aldehydes, wedge bonds, carbon triple bonds or hetero atoms are retained.
- *keepwedge*
  keep hydrogens which are at the end of a wedge bond, indicating stereochemistry.
- *normalize*
  Normalize the wedge pattern for standard cases, removing wedges from hydrogens if the result is still stereochemically defined. Hydrogens which lose their wedge in this process are no longer protected by the *keepwedge* flag.
- *wedgetransfer*
  If a hydrogen atom is removed which is at the end of a wedge, the wedge information is saved by transferring the wedge (changing its up/down status if necessary) to an adjacent, surviving bond. This flag has no effects if the *keepspecial* or *keepwedge* flags are set. This flag is set by default.

If the *flags* parameter is an empty string, or *none*, it is ignored. The default flag value is *wedgetransfer* - but the default value is overridden if any flags are set!

The return value is the total number of hydrogens deleted.

A **PYTHON** implementation would not be useful because all applicable major objects have a **`hstrip()`** method, which can be invoked without exact knowledge of the object class.

majorobj jget

```
majorobj jget handle propertylist ?filterset? ?parameterdict?
```

This is a variant of **`majorobj get`** which returns the result data as a **JSON** formatted string instead of **TCL** or **PYTHON** interpreter objects. The command is usable only for property data, not attribute retrieval.

A **PYTHON** implementation would not be useful because every major object has a **`jget()`** method, which can be invoked without exact knowledge of the object class.

majorobj jnew

`majorobj jnew` *`handle propertylist ?filterset? ?parameterdict?`*

This is a variant of **`majorobj new`** which returns the result data as a **JSON** formatted string instead of **TCL** or **PYTHON** interpreter objects.

A **PYTHON** implementation would not be useful because every major object has a **`jnew()`** method, which can be invoked without exact knowledge of the object class.

majorobj jshow

`majorobj jshow` *`handle propertylist ?filterset? ?parameterdict?`*

This is a variant of **`majorobj show`** which returns the result data as a **JSON** formatted string instead of **TCL** or **PYTHON** interpreter objects.

A **PYTHON** implementation would not be useful because every major object has a **`jshow()`** method, which can be invoked without exact knowledge of the object class.

majorobj ldup

`majorobj ldup` *`?handlelist?...`*
`Majorobj.ldup(?mref/mrefsequence/handle?,...)`

Duplicate all objects in the handle list(s) in default mode.

The return value is a single list (even if multiple source lists are used) of the duplicated object handles or references. If an argument list element is an empty string (or **`None`** for **PYTHON**), it indicates a missing object, and the output list also receives an empty string (or **`None`**) element at its position, without raising an error.

majorobj lhdup

`majorobj lhdup` *`?handlelist?...`*
`Majorobj.lhdup(?mref/mrefsequence/handle?,...)`

Duplicate all objects in the handle list(s) in default mode, and add hydrogens if applicable (ensembles, reactions, datasets).

The return value is a single list (even if multiple source lists are used) of the duplicated object handles. If an argument list element is an empty string (or **`None`** for **PYTHON**), it indicates a missing object, and the output list also receives an empty string (or **`None`**) element at its position, without raising an error.

majorobj new

`majorobj new` *`handle propertylist ?filterset? ?parameterdict?`*

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **`majorobj get`** command. The difference between **`majorobj get`** and **`majorobj new`** is that the latter forces the re-computation of the property data, regardless whether it is present and valid, or not.

A **Python** implementation would not be useful because every major object has a **`new()`** method, which can be invoked without exact knowledge of the object class.

majorobj nget

```
majorobj nget handle propertylist ?filterset? ?parameterdict?
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **`majorobj get`** command. The difference between **`majorobj get`** and **`majorobj nget`** is that the latter always returns numeric data, even if symbolic names for the values are available.

A **Python** implementation would not be useful because every major object has a **`nget()`** method, which can be invoked without exact knowledge of the object class.

majorobj nnew

```
majorobj nget handle propertylist ?filterset? ?parameterdict?
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **`majorobj get`** command. The difference between **`majorobj get`** and **`majorobj nget`** is that the latter always returns numeric data, even if symbolic names for the values are available.

For examples, see the **`majorobj get`** command. The difference between **`majorobj get`** and **`majorobj nnew`** is that the latter always returns numeric data, even if symbolic names for the values are available, and that property data re-computation is enforced.

A **Python** implementation would not be useful because every major object has a **`nnew()`** method, which can be invoked without exact knowledge of the object class.

majorobj show

```
majorobj show handle propertylist ?filterset? ?parameterdict?
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **`majorobj get`** command. The difference between **`majorobnj get`** and **`majorobj show`** is that the latter does not attempt computation of property data, but raises an error if the data is not present and valid. For data already present, **`majorobj get`** and **`majorobj show`** are equivalent.

A **Python** implementation would not be useful because every major object has a **`show()`** method, which can be invoked without exact knowledge of the object class.

majorobj sqldget

`majorobj sqldget` *`nhandle propertylist ?filterset? ?parameterdict?`*

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **`majorobj get`** command. The differences between **`majorobj get`** and **`majorobj sqldget`** are that the latter does not attempt computation of property data, but initializes the property value to the default and returns that default, if the data is not present and valid; and that the **SQL** command variant formats the data as **SQL** values rather than for **TCL** or **PYTHON** script processing.

A **PYTHON** implementation would not be useful because every major object has a **`sqldget()`** method, which can be invoked without exact knowledge of the object class.

majorobj sqlget

`majorobj sqlget` *`nhandle propertylist ?filterset? ?parameterdict?`*

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **`majorobj get`** command. The difference between **`majorobj get`** and **`majorobj sqlget`** is that the **SQL** command variant formats the data as **SQL** values rather than for **TCL** or **PYTHON** script processing.

A **PYTHON** implementation would not be useful because every major object has a **`sqlget()`** method, which can be invoked without exact knowledge of the object class.

majorobj sqlnew

`majorobj sqlnew` *`nhandle propertylist ?filterset? ?parameterdict?`*

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **`majorobj get`** command. The differences between **`majorobj get`** and **`majorobj sqlnew`** are that the latter forces re-computation of the property data, and that the **SQL** command variant formats the data as **SQL** values rather than for **TCL** or **PYTHON** script processing.

A **PYTHON** implementation would not be useful because every major object has a **`sqlnew()`** method, which can be invoked without exact knowledge of the object class.

majorobj sqlshow

`majorobj sqlshow` *`nhandle propertylist ?filterset? ?parameterdict?`*

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **`majorobj get`** command. The differences between **`majorobj get`** and **`majorobj sqlshow`** are that the latter does not attempt computation of property data, but raises an error if the data is not present and valid, and that the **SQL** command variant formats the data as **SQL** values rather than for **TCL** or **PYTHON** script processing.

A **PYTHON** implementation would not be useful because every major object has a **`sqlshow()`** method, which can be invoked without exact knowledge of the object class.

majorobj subcommands

```
majorobj subcommands
dir(Majorobj)
```

List all currently implemented subcommands of this command.

The **PYTHON** version is somewhat misleading, because it also lists methods which are implemented in a generic fashion for all major objects and then inherited by specialized classes.

majorobj valid

```
majorobj valid handle propertylist
```

Returns a list of boolean values indicating whether values for the named properties are currently set for the object. No attempt at computation is made.

`majorobj has` is an alias to this command.

A **PYTHON** implementation would not be useful because every major object has a **`valid()`** method, which can be invoked without exact knowledge of the object class.

The *mol* Command

The **mol** command is the generic command used to manipulate molecules. The syntax of this command follows the standard schema of *command/subcommand/majorhandle/minorlabel*.

The **mol** command supports, in addition to the normal label decoding process, the magic value *primary* as molecule label. The primary molecule is determined, in this order, by the maximum value of properties `M_HEAVY_ATOM_COUNT`, `M_NATOMS` and `M_HASHISY`. When there is more than one molecule where all properties are top-rated, the first molecule of these in the molecule list is chosen. An empty ensemble has no primary molecule. The pseudo molecule labels *first*, *last* and *random* are additional special values, which select the first molecule in the molecule list, the last, or a random molecule.

Examples:

```
mol get $ehandle 1 M_WEIGHT
mol delete $ehandle 2
mol dup $ehandle primary
set pmol_label [mol mol $ehandle primary]
```

This is the list of officially supported subcommands:

mol append

```
mol append ehandle label ?property value?...
m.append({?property:value,?...})
m.append(?property,value,?...)
```

Standard data manipulation command for appending property data. It is explained in more detail in the section about setting property data.

The command returns the first data value.

Example:

```
mol append $ehandle 1 M_NAME "_linker"
```

mol align3d

```
mol align3d ehandle label box/center/masscenter/pmi ?usehydrogens? ?property?
m.align3d(?mode=?,?usehydrogens=?,?coordinateproperty=?)
```

Perform a 3D alignment by modifying standard atom coordinates property `A_XYZ`, or an alternative explicitly specified atomic coordinate property.

The possible alignment modes are

- *box*
  move center of enclosing 3D coordinate box to origin
- *center*
  move average atom coordinates to origin
- *masscenter*
  move mass-weighted atom coordinates to origin

- *pmi*
  align ensemble to principle moments of inertia (largest on x axis), and move the mass-weighted center to the origin.

By default all atoms of the molecule are used to compute the alignment rotation and movement vectors, including hydrogens. If these should be omitted from computing the movement vectors (but not the subsequent atom movement), the optional *usehydrogens* parameter can be set to *false*.

The command returns the label or reference of the molecule.

mol atoms

```
mol atoms ehandle label ?filterset? ?filtermode?
m.atoms(?filters=?,?mode=?)
```

Standard cross-referencing command to obtain the labels or references of the atom in the molecule. This is explained in more detail in the section about object cross-references.

Example:

```
mol atoms $ehandle !hydrogen
```

returns the labels of the non-hydrogen atoms in the molecule.

mol bondangles

```
mol bondangles ehandle label ?filterset? ?filtermode?
m.bondangles(?filters=?,?mode=?)
```

Standard cross-referencing command to obtain the labels or references of the bond angle objects the molecule contains. This is explained in more detail in the section about object cross-references.

mol bonds

```
mol bonds ehandle label ?filterset? ?filtermode?
m.bonds(?filters=?,?mode=?)
```

Standard cross-referencing command to obtain the labels or references of the bonds the molecule contains. This is explained in more detail in the section about object cross-references. Bonds which cross into other molecules are not listed. Such bonds may exist if they are not of a bond type which is used to group atoms into molecules.

Examples:

```
mol bonds $ehandle 1
mol bonds $ehandle 1 {1 doublebond triplebond} count
```

The first example returns all labels of the bonds molecule 1 contains. The second example returns the number of double or triple bonds in the molecule.

mol compare

```
mol compare ehandle label ehandle2 label2
m.compare(mref)
```

Compare two molecules, yielding a stable sort order. The compared attributes are, in this order, the number of atoms, the number of bonds, the molecular weight, the number of ESSSR rings and finally the stereo- and isotope aware 64-bit hashcode (`M_ISOTOPE_STEREO_HASHY`). The command returns 1 if the first molecule is larger, -1 if the second is larger, and 0 if they are identical according to the comparison scheme.

The compared property values, with the exception of the final hashcode tiebreaker, are compatible with the **RDKit** model.

mol defined

```
mol defined ehandle label property
m.defined(property)
```

This command checks whether a property is defined for the molecule. This is explained in more detail in the section about property validity checking. Note that this is *not* a check for the presence of property data! The **`ens valid`** command is used for this purpose.

Example:

```
mol defined $ehandle 1 M_NAME
```

checks whether molecule 1 is of a type for which `M_NAME` is defined.

mol delete

```
mol delete ehandle ?label?...
mol delete ehandle all
m.delete()
Mol.Delete(eref,"all")
Mol.Delete(mref,...)
Mol.Delete(eref,?mlabel/mref/mrefsequence?,...)
```

Delete molecules from the ensemble. All minor objects on the same ensemble which contain atoms from the deleted molecules, such as rings, groups and ring systems, are also deleted. However, these minor object sets are not completely destroyed and property data on the remaining objects remains valid, if those properties are not invalidated by *merge* events.

Deleting a molecule triggers a *merge* invalidation event, but not *atomchange/bondchange* events. Property data which is susceptible to this invalidation mode is recursively deleted from the ensemble.

The special label *all* deletes all molecules in the ensemble. Usually this is equivalent to **`ens clear`**, but in theory there may exist atom-class objects which are not part of a molecule, and these are then retained.

The command returns the number of deleted molecules.

Example:

```
mol delete $ehandle 1
```

mol dget

```
mol dget ehandle label propertylist ?filterset? ?parameterdict?
m.dget(property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **`mol get`** command. The difference between **`mol get`** and **`mol dget`** is that the latter does not attempt computation of property data, but rather initializes the property values to the default and return that default if the data is not yet available. For data already present, **`mol get`** and **`mol dget`** are equivalent.

mol dup

```
mol dup ehandle label ?datasethandle? ?position?
m.dup(?target=?,?position=?)
```

Duplicate a single molecule into a new ensemble. The function returns the new ensemble handle or reference.

By default, the new ensemble is appended to the same dataset as the original ensemble, or placed outside of any dataset if the input ensemble was not a dataset member. If the optional dataset handle parameter is specified, the duplicate is directly moved to that dataset. If an empty string (or **`None`** for **PYTHON**) is passed, the duplicate is not made a dataset member, even if the input ensemble is in a dataset.

The new ensemble preserves ring information and associated property data and other minor object data from the original ensemble for all minor objects which exclusively refer to atoms which are part of the duplicated molecule. Minor objects outside the duplicated molecule, or spanning multiple molecules are not duplicated.

Example:

```
mol dup $ehandle 1 [dataset create]
```

Duplicate the molecule with label one and move the new single-molecule ensemble into a newly created dataset.

mol ens

```
m.ens()
```

PYTHON-only method to get the ensemble reference from a molecule reference.

mol exists

```
mol exists ehandle label ?filterlist?
m.exists(?filters=?)
Mol.Exists(eref,label,?filters=?)
```

Check whether this molecule exists. Optionally, a filter list can be supplied to check for the presence of specific features. The command returns 0 if the molecule does not exist, or fails the filter, and 1 in case of successful testing.

Example:

```
mol exists $ehandle 99
```

mol expand

```
mol expand ehandle label ?allowambiguous? ?noimplicith?
m.expand(?allowambiguous=?,?noimplicith=?)
```

This command expands all superatoms in the molecule. The mechanisms for the expansion of superatoms are described in detail for the **`atom expand`** command. This command is functionally equivalent, working on all atoms in the molecule instead a single atom.

Example:

```
mol expand $ehandle 1
```

The command returns the total number of successfully expanded atoms.

mol expr

```
mol expr ehandle label expression
m.expr(expression)
```

Compute a standard **SQL**-style property expression for the molecule. This is explained in detail in the chapter on property expressions.

mol fill

```
mol fill ehandle label ?property value?...
m.fill({property:value,...})
m.fill(?property,value?,...)
```

Standard data manipulation command for setting data, ignoring possible mismatches between the lengths of the lists of objects associated with the property and the value list. It is explained in more detail in the section about setting property data.

The command returns the first fill value.

Example:

```
mol fill $ehandle 1 B_COLOR red
```

sets the color of the first bond molecule 1 contains to *red*.

mol filter

```
mol filter ehandle label filterlist
m.filter(filters)
```

Check whether a molecule passes a filter list. The boolean return value is 1 for success and 0 for failure.

Example:

```
mol filter $ehandle 1 [list carbon doublebond]
```

checks whether the molecule contains one or more carbon atoms and one or more double bonds. The double bond does not need to be with a carbon atom.

mol get

```
mol get ehandle label propertylist ?filterset? ?parameterdict?
m.get(property=,?filters=?,?parameters=?)
m[property]
m.property
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

Examples:

```
mol get $ehandle 1 {M_WEIGHT A_ELEMENT}
```

yields a list with two elements, consisting of the molecular weight as the first element and the element numbers of all atoms in the molecule as a nested list as the second result list element. If the information is not yet available, an attempt is made to compute it. If the computation fails, an error results.

```
mol get $ehandle 1 B_ORDER ringbond
```

gives the bond orders of all bonds of the molecule which are ring bonds.

For the use of the optional property parameter list argument, refer to the documentation of the **`ens get`** command.

Variants of the *mol get* command are **`mol new`**, **`mol dget`**, **`mol show`**, **`mol sqldget`**, **`mol sqlget`**, **`mol sqlnew`** and **`mol sqlshow`**.

Further examples:

```
mol get $ehandle 1 E_NAME
mol get $ehandle 1 A_FLAGS(boxed)
```

mol groups

```
mol groups ehandle label ?filterset? ?filtermode?
m.groups(?filters=?,?mode=?)
```

Standard cross-referencing command to obtain the labels or references of the groups the molecule contains. This is explained in more detail in the section about object cross-references. Groups which contain atoms from more than one molecule are included.

Example:

```
mol groups $ehandle 1
```

mol hadd

```
mol hadd ehandle label ?filterset? ?flags?
m.hadd(?filters=?,?flags=?)
```

Add a standard set of hydrogens to the molecule. If the *filterset* parameter is specified, only those atoms which pass the filter set are processed.

Additional operation flags may be activated by setting the *flags* parameter to a list of flag names, or a numerical value representing the bit-ored values of the selected flags. By default, the flag set is empty, corresponding to the use of an empty string or *none* as parameter value. These flags are currently supported:

- *keepflags*
  For expert use only. Do not discard min/max values and property scope flags for atom properties when hydrogen is added.
- *no2dcoords*
  Do not assign 2D coordinates to the added hydrogens, even if the rest of the atoms in the ensemble have valid 2D coordinates. In any case, 2D coordinates are never added when the ensemble does no already possess valid 2D coordinates.

- *no3dcoords*
  Do not assign 3D coordinates to the added hydrogens, even if the rest of the atoms in the ensemble have valid 3D coordinates. In any case, 3D coordinates are never added when the ensemble does no already possess valid 3D coordinates.
- *noanions*
  Do not add hydrogen to atoms with a negative formal charge.
- *noatoms*
  Do not add hydrogen to atoms without any bonds.
- *nocations*
  Do not add hydrogen to atoms with a positive formal charge.
- *noelements*
  Do not add hydrogen if the ensemble consists purely of isolated metal atoms, which probably represent the material in elementary form, or as an alloy.
- *noexcessvalences*
  Similar to *nohighvalences*, but hydrogen is not added to any atom which is not in its lowest standard bonded valence state.
- *nofixatomtext*
  Do not adjust property `A_TEXTLABEL` (if present) by removing references to implicit H from it on atoms where hydrogen is added. For example, by default “NHCOOEt” becomes “NCOOEt” after adding an instantiated hydrogen to the nitrogen atom. This reduces confusion on the hydrogen status when rendering all atoms.
- *nohighvalences*
  Do not add hydrogen to atoms which already exceed their lowest standard valence minus any formal charge. This option only applies to elements which have a defined lowest standard valence (this is configurable via the element table).
- *nomemory*
  Do not remember the added hydrogen atoms as automatically added. Normally, a flag is retained as part of the atom information which distinguishes atoms which were added by automatic processing, such as hydrogen addition, from those which were originally input.
- *nometals*
  Do not attempt to add hydrogen to atoms which are metals (as defined in the system element table).
- *nospecial*
  Do not perform hydrogen addition to atoms which participate in non-standard bonds (all bonds with `B_TYPE` not *normal*).
- *protonate*
  Add a single proton to the molecule, to the first suitable atom. The charge of the selected atom is increased, only a single hydrogen is added regardless of the standard number of missing hydrogens, and this command *will* issue the standard property invalidation event for atom and bond changes. In the molecule command variant, this option is rarely useful. It is supported for compatibility with the **`atom hadd`** command.
- *resetmemory*
  Reset the origin flag described above for all atoms in the ensemble. All current atoms appear to be part of the original atom set.

Adding hydrogens with this command, except if the *protonate* flag is set, is less destructive to the property data set of the ensemble than adding them with individual **`atom create/bond create`** commands, because many properties are designed to be indifferent to explicit hydrogen status changes, but are invalidated if the structure is changed in other ways.

The command returns the number of hydrogens which were added.

Example:

```
set ehandle [ens create {[C].[C]}]
mol hadd $ehandle 1
```

adds four hydrogens to the first carbon atom, transforming it into methane, but leave the second carbon atom untouched.

mol hdup

```
mol hdup ehandle label ?datasethandle? ?position?
m.hdup(?target=?,?position=?)
```

This command provides the same functionality as **`mol dup`**, except that it also adds a standard set of hydrogens to the new ensemble.

mol hstrip

```
mol hstrip ehandle label ?flags?
m.hstrip(?flags=?)
```

This command removes hydrogens from the selected molecule. By default, all hydrogen atoms in the molecule are removed.

The *flags* parameter can be used to make the operation more selective. It may be a list of the following flags:

- *deprotonate*
  If this flag is set, a single proton is removed from the first suitable atom. This command variant triggers a standard atom and bond change property invalidation event, and it always ends processing after removing the first proton. Proton removal decreases the charge of the atom by one. In the molecule command variant, this flag is rarely useful - it is supported for compatibility with the **`atom hstrip`** command.
- *keepalphawedge*
  Keep hydrogen atoms which are bonded to an atom which is at the tip of a wedgebond. This flag excludes the case where the bond to the hydrogen atom is the wedge bond - use the *keepwedge* flag to cover this case.
- *keepisotopes*
  Keep hydrogen atoms which are isotope labels (including enriched/depleted ^{1}H).
- *keeporiginal*
  Hydrogen atoms which were not automatically added via a hydrogen addition command are retained. Note these commands commands can be run in a mode which does not leave information about automatic addition - hydrogens added this way are not protected.

- *keepprotons*
  Keep any molecules which consist only of hydrogen atoms (such as protons, hydride anions, and molecular hydrogen).
- *keepspecial*
  If this flag is set, hydrogens which are usually displayed, such as on aldehydes, wedge bonds, carbon triple bonds or hetero atoms are retained.
- *keepwedge*
  Keep hydrogens which are at the end of a wedge bond, indicating stereochemistry.
- *normalize*
  Normalize the wedge pattern for standard cases, removing excess wedges from hydrogens if the result structure is still stereochemically defined. Hydrogens which lose their wedge in this process are no longer protected by the *keepwedge* flag.
- *wedgetransfer*
  If a hydrogen atom is removed which is at the end of a wedge, the wedge information is saved by transferring the wedge (changing its up/down status if necessary) to an adjacent, surviving bond. This flag has no effects if the *keepspecial* or *keepwedge* flags are set. This flag is set by default.

If the *flags* parameter is an empty string, or *none*, it is ignored. The default flag value is *wedgetransfer* - but the default value is overridden if any flags are set!

Hydrogen stripping is not as disruptive to the ensemble data content as normal atom deletion, except when the *deprotonate* flag is set. The system assumes that this operation is done as part of some file output or visualization preparation. However, if any new data is computed after stripping, the computation functions see the stripped structure, and proceed to work on that reduced structure without knowledge that the structure may contain implicit hydrogens.

The return value of the command is the number of hydrogens removed.

Example:

```
mol hstrip $ehandle 1 [list keeporiginal wedgetransfer]
```

mol hydrogenate

```
mol hydrogenate ehandle label ?filterset? ?changeset?
m.hydrogenate(?filters=?,?changeset=?)
```

Reduce all bonds in the molecule to single bonds except those excluded by the filter set.

If a change set is supplied, its interpretation is the same as in **`mol hadd`**.

The command returns the number of added hydrogens.

Example:

```
mol hydrogenate $eh 1 {!arobond !ccbond}
```

This reduces all non-aromatic hetero bonds in molecule 1 to single bonds.

mol image

```
mol image ehandle label ?width? ?height? ?options?
```

This command generates a **Tk** image object displaying the molecule as an icon. The command is only available in toolkit variants which are linked with the portable **Tk GUI** toolkit library and which are either statically linked with the **GD** image drawing library, or can load it dynamically. It is currently not support in the **Python** interface.

The default image size is 64x64 pixels, but this may be overridden by the *width* and *height* parameters. If only *width* is set, it is also used for the height. The command returns a Tk image handle. These images may for example be placed on Tk canvases as canvas objects, or used on buttons and other GUI objects.

Because of the small size of the images, atoms are not displayed as symbols, but small color-coded squares. This is a command for the implementation of graphical structure-handling applications with icons. For serious structure visualization, use the `E_GIF`, `E_EMF_IMAGE` or `E_EPS_IMAGE` properties.

Additional options may be added by an arbitrary sequence of option/value pairs. Color names can be those registered in the **X11** color database, or a numeric specification in the *#rrggbb* format. These options are currently supported:

- -background *color*
  Background color. The default is *black*.
- -border *npixels*
  Thickness of the image border. The default are 5 pixels.
- -bordercolor *color*
  Border color. The default is *blue*.
- -cmode *none/special/all*
  Display mode for carbon atoms. The default is *special*, meaning that only carbon atoms which usually are drawn with a C symbol are displayed as colored rectangle and not just a bond node. Highlighted atoms are always displayed.
- -highlightatom *label*
  Select an atom for highlighting. By default, no atom is highlighted.
- -highlightcolor *color*
  Set the highlighting color. The default is *chartreuse*.
- -hmode *none/special/all*
  Display mode for hydrogen atoms. The default is *special*, meaning that only hydrogen atoms which usually are drawn with an H symbol are displayed as colored rectangle. Other hydrogen atoms and the bonds leading to them are suppressed. Highlighted atoms are always displayed.
- -imagename *name*
  Explicitly set a name for the image. By default, a name of the form *image**n*** is automatically generated. It is possible to specify the name of an existing image, which will then be overwritten.
- -linecolor *color*
  Color of bond lines and wedges. The default is *white*.

Images are cached. If an image for the selected molecule with the same display attributes exists, it is reused.

Example:

```
set img [mol image $ehandle 1 80 80 -border yellow -linecolor blue]
canvas create .canvaswin image 50 50 -image $img
```

mol index

```
mol index ehandle label
m.index()
```

Get the index of the molecule. The index is the position in the molecule list of the ensemble. The first position is index 0.

Example:

```
mol index $ehandle 99
```

mol isotopecheck

```
mol isotopecheck ehandle label ?failedatomvariable? ?extended?
m.isotopecheck(variable=,extended=)
```

Test whether the isotope labels on the atoms of the molecule, if they exist, are physically reasonable. The command returns the number of failed atoms. If a capture variable is specified, the atom labels or references of these atoms are stored therein. If no isotope labels are set in `A_ISOTOPE`, the command always reports zero problems.

By default, a smaller isotope table is used which contains only isotopes which are sufficiently long-lived to perform chemistry on. These include naturally occurring isotopes as well as isotopes used for experimental labeling, such as ^{3}H or ^{14}C. If the *extended* boolean flag is set, a larger table containing all known isotopes of the elements is used.

The *isocheck* command is an alias.

mol jget

```
mol jget ehandle label propertylist ?filterset? ?parameterdict?
m.jget(property=,?filters=?,?parameters=?)
```

This is a variant of **`mol get`** which returns the result data as a **JSON** formatted string instead of **TCL** or **PYTHON** interpreter objects.

mol jnew

```
mol jnew ehandle label propertylist ?filterset? ?parameterdict?
m.jnew(property=,?filters=?,?parameters=?)
```

This is a variant of **`mol new`** which returns the result data as a **JSON** formatted string instead of **TCL** or **PYTHON** interpreter objects.

mol jshow

```
mol jshow ehandle label propertylist ?filterset? ?parameterdict?
m.jshow(property=,?filters=?,?parameters=?)
```

This is a variant of **`mol show`** which returns the result data as a **JSON** formatted string instead of **TCL** or **PYTHON** interpreter objects.

mol local

```
mol local ehandle label propertylist ?filterset? ?parameterdict?
m.local(property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading and recalculating object data. It is explained in more detail in the section about retrieving property data.

Example:

```
mol local $ehandle 1 A_LABEL_STEREO
```

Note that very few computation routines currently support the local re-computation of data - in most cases, this command falls back to a global re-computation.

mol match

```
mol match ehandle label ss_ehandle ?ss_label? ?matchflags? ?ignoreflags?
   ?atommatchvar? ?bondmatchvar? ?molmatchvar?
m.match(substructure=,?substructuremol=?,?matchflags=?,?ignoreflags=?,
   ?atommatchvariable=?,?bondmatchvariable=?,?molmatchvariable=?)
```

Check whether the selected molecule matches a substructure. Only the first molecule, or the molecule selected by the substructure label parameter, is tested. The substructure may be part of any structure ensemble, and even be in the same ensemble as the primary command molecule. Both the atoms in the molecule and the bonds between them are checked.

The precise operation of the substructure match routine can be tuned by providing a standard set of match flags and feature ignore flags. The default match flag set has set bits for the *bondorder*, *atomtree* and *bondtree* comparison features, and an empty ignore set. If a flag set is specified as an empty string, the default set is used. In order to reset a flag set, an explicit *none* value must be used. The bit options of the match flag are explained in the documentation of the **`match ss`** command.

The command returns 1 for a successful match, 0 otherwise. If an optional atom, bond, or molecule match variable is specified, it is set to a nested list of matching substructure/structure atom, bond or molecule labels. If no match can be found, the variable is set to an empty list. In case only a bond or molecule match variable is needed, an empty string can be used to skip the unused match variable argument positions.

Example:

```
set ss [ens create {c1ccccc1.c1ncccc1} smarts]
set m_contains_phenylring [mol match $ehandle $label $ss 1]
```

mol mol

```
mol mol ehandle label
Mol.Ref(eref,identifier)
```

Standard cross-referencing command to obtain the label or reference of the molecule as stored in property `M_LABEL`. This is explained in more detail in the section about object cross-references.

Example:

```
mol mol $ehandle #0
```

returns the label of the first molecule of the ensemble molecule list.

```
mol mol $ehandle primary
```

returns the label of the primary molecule in the ensemble.

mol new

```
mol new ehandle label propertylist ?filterset? ?parameterdict?
m.new(property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **`mol get`** command. The difference between **`mol get`** and **`mol new`** is that the latter forces the re-computation of the property data, regardless whether it is present and valid, or not.

mol nget

```
mol nget ehandle label propertylist ?filterset? ?parameterdict?
m.nget(property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **`mol get`** command. The difference between **`mol get`** and **`mol nget`** is that the latter always returns numeric data, even if symbolic names for the values are available.

mol pack

```
mol pack ehandle label ?maxsize? ?requestprops? ?suppressedprops? ?compressionlib?
m.pack(?maxsize=?,?requestprops=?,?suppressedprops=?,?compressionlib=?)
```

Pack the selected molecule into a base64-encoded compressed serialized object string. This string does not contain any non-printable characters and is a full dump of the internal state of the object, omitting only property data that was declared to be so easily re-computed that a dump is not worthwhile. Outside object relationship information, such as the dataset the ensemble of the molecule might be a member of, or associated tables of the parent ensemble are not included, and neither are any ensemble properties.

The maximum size of the object string (default -1, meaning unlimited) can be configured by the optional *maxsize* parameter. The size is specified in bytes. If the pack string would be longer than the maximum size, an error results.

The next two optional parameters allow to request a specific property set to be part of the package, even if it normally would not be included, and to explicitly omit properties from the dump. No property computation is performed, and suppressed properties are not purged from the ensemble.

The default compression library is *zlib*. Other useful variants include *lzo* and *gzip* (and there are other internal types), but these may not be available on all builds due to license issues, and you need to specify the compression library when a dataset is unpacked. It is generally recommended to stay with *zlib*.

Single-molecule ensembles can be restored from a packed object string by the **`ens unpack`** and **`ens create`** commands.

The **PYTHON** **`Mol`** class also supports the pickle protocol, but not unpickling. Restoring a pickled molecule can be done via the **`Ens`** unpickle method - this is like unpacking the pack string, which also returns a new single-mol ensemble.

mol pis

```
mol pis ehandle label ?filterset? ?filtermode?
m.pis(filters=?,?mode=?)
```

Standard cross-referencing command to obtain the labels or references of the π systems the molecule contains. This is explained in more detail in the section about object cross-references.

Examples:

```
mol pis $ehandle 1
```

π systems are a rather exotic feature and not commonly used. These are essentially descriptions of bonding interactions which use p or d orbitals, such as in standard covalent multiple bonds. A simple double bond is described with one σ system and one π system in this representation.

mol ref

```
Mol.Ref(eref,identifier)
```

PYTHON only method to get a group reference. See **`mol mol`** command.

mol replicate

```
mol replicate ehandle label ?count?
m.replicate(?count=?)
```

Add duplicates of the selected molecule to the command ensemble. The default number of duplicates is one, but any other number may be requested by setting the *count* parameter. If the count is less than one, the command is silently ignored.

The command returns the labels or references of all newly created molecules as a list. New molecule labels begin at one plus the highest old label. All atoms, bonds and other chemistry objects which are created by the command are appended to the object lists in the ensemble and will thus bear higher labels than any existing label of their class in the ensemble. This command triggers a *merge* property invalidation event.

The **`mol dup`** command duplicates a molecule into a new ensemble, which is quite different from what this command does.

Example:

```
set eh [ens create C.CC]
mol dup $eh 1 2
echo [ens get $eh E_SMILES]
```

returns *C.CC.C.C*, because the first molecule (label one) was duplicated twice.

mol rings

```
mol rings ehandle label ?filterset? ?filtermode?
m.rings(?filters=?,?mode=?)
```

Standard cross-referencing command to obtain the labels or references of the rings the molecule contains. This is explained in more detail in the section about object cross-references. Rings which are not restricted to the selected molecule are listed. Under certain circumstances, it is possible to have rings which span more than one molecule.

Examples:

```
mol rings $ehandle 1
mol rings $ehandle 1 [list heterocycle aroring]
```

The first example returns the labels of all rings the molecule contains. If the molecule does not contain any rings, an empty list is returned. Only labels of rings in the **SSSR** or **ESSSR** set are returned, even if the currently computed ring set is larger. The second example filters the rings - only heteroaromatic rings are reported.

mol ringsystems

```
mol ringsystems ehandle label ?filterset? ?filtermode?
m.ringsystems(?filters=?,?mode=?)
```

Standard cross-referencing command to obtain the labels or references of the ring systems the molecule contains. This is explained in more detail in the section about object cross-references. Ring systems which are not restricted to the selected molecule are included. Under certain circumstances, it is possible to have ring systems which span more than one molecule.

Examples:

```
mol ringsystems $ehandle 1
mol ringsystems $ehandle 1 [list heterocycle aroring]
```

The first example returns the labels of all ring systems the molecule contains. If the molecule does not contain any ring systems, an empty list is returned. The second example filters the ring systems - a ring system label is included in the output list only if that ring system contains one or more hetero aromats.

mol rotate

```
mol rotate ehandle label angle axis ?center? ?property?
m.rotate(angle=,axis=,?center=?,?coordinateproperty=?)
```

Rotate the molecule in 3D space on property `A_XYZ` or a custom atom float vector coordinate property.

This command requires 3D atomic coordinates. If these are not yet present, an attempt is made to compute them. The rotation angle is specified in degrees. The first point is the axis vector - it can be specified in any format the **Tcl vec** command understands. By default the center of rotation is the center of the molecule bounding box, but by providing explicit center coordinates, any center of rotation can be set.

This operation triggers a *3Dop* property invalidation event.

The command returns the original molecule label or reference.

Example:

```
mol rotate $ehandle 1 50 {1 1 0}
```

rotates the molecule around its center 50 degrees counter-clockwise along the 45-degrees xy-diagonal.

mol set

```
mol set ehandle label ?property value?...
m.set(?property,value?,...)
m.set({property:value,...})
```

```
m.property = value
m[property] = value
```

Standard data manipulation command for setting property data. It is explained in more detail in the section about setting property data.

The command returns the first data value.

Example:

```
mol set $ehandle 1 M_NAME "Pharmacon X-25"
```

mol show

```
mol show ehandle label propertylist ?filterset? ?parameterdict?
m.show(property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **`mol get`** command. The difference between **`mol get`** and **`mol show`** is that the latter does not attempt computation of property data, but raises an error if the data is not present and valid. For data already present, **`mol get`** and **`mol show`** are equivalent.

mol sigmas

```
mol sigmas ehandle label ?filterset? ?filtermode?
m.sigmas(?filters=?,?mode=?)
```

Standard cross-referencing command to obtain the labels or references of the σ systems the molecule contains. This is explained in more detail in the section about object cross-references.

Examples:

```
mol sigmas $ehandle 1
```

σ systems are a rather exotic feature and not commonly used. These are essentially descriptions of bonding interactions which use s orbitals, such as normal, covalent single bonds, or the central bond in multiple bonds. A simple double bond is described with one σ system and one π system in this representation.

mol sqldget

```
mol sqldget ehandle label propertylist ?filterset? ?parameterdict?
m.sqldget(property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **`mol get`** command. The differences between **`mol get`** and **`mol sqldget`** are that the latter does not attempt computation of property data, but initializes the property value to the default and returns that default, if the data is not present and valid; and that the **SQL** command variant formats the data as **SQL** values rather than for **Tcl** or **Python** script processing.

mol sqlget

```
mol sqlget ehandle label propertylist ?filterset? ?parameterdict?
m.sqlget(property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **`mol get`** command. The difference between **`mol get`** and **`mol sqlget`** is that the **SQL** command variant formats the data as **SQL** values rather than for **Tcl** or **Python** script processing.

mol sqlnew

```
mol sqlnew ehandle label propertylist ?filterset? ?parameterdict?
m.sqlnew(property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **`mol get`** command. The differences between **`mol get`** and **`mol sqlnew`** are that the latter forces re-computation of the property data, and that the **SQL** command variant formats the data as **SQL** values rather than for **Tcl** or **Python** script processing.

mol sqlshow

```
mol sqlshow ehandle label propertylist ?filterset? ?parameterdict?
m.sqlshow(property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **`mol get`** command. The differences between **`mol get`** and **`mol sqlshow`** are that the latter does not attempt computation of property data, but raises an error if the data is not present and valid, and that the **SQL** command variant formats the data as **SQL** values rather than for **Tcl** or **Python** script processing.

mol subcommands

```
mol subcommands
dir(Mol)
```

Lists all subcommands of the **`mol`** command. Note that this command does not require an ensemble handle, or a molecule label.

mol surfaces

```
mol surfaces ehandle label ?filterset? ?filtermode?
m.surfaces(?filters=?,?mode=?)
```

Standard cross-referencing command to obtain the labels or references of surface patches the molecule is associated with. This is explained in more detail in the section about object cross-references.

Example:

```
mol surfaces $ehandle $label
```

Note that surface patches are not required to be associated with any atom, and if they are not, they are implicitly not associated with any molecule.

mol torsions

```
mol torsions ehandle label ?filterset? ?filtermode?
m.torsions(?filters=?,?mode=?)
```

Standard cross-referencing command to obtain the labels or references of the torsion angle objects the molecule contains. This is explained in more detail in the section about object cross-references.

mol translate

```
mol translate ehandle label pt1 ?pt2? ?property?
m.translate(point1=,?point2=?,?coordinateproperty=?)
```

Move the atoms of the molecule by modifying their 3D coordinates in property `A_XYZ`, or a custom atomic float vector coordinate property. This command requires atomic 3D coordinates and will attempt to compute them if they are not yet present. If no 3D atomic coordinates can be generated, the command fails with an error.

The movement vector may either be specified by a single vector, or two points. If two points are used, the subtraction of the second point from the first is used to compute the movement vector. Both point/vector arguments understand the same vector notation syntax as the **`vec`** command.

This command triggers a *3Dglop* property invalidation event.

The command returns the original molecule label or reference.

Example:

```
mol translate $ehandle 1 {1 0 0} {2 0 0}
```

moves the molecule one Ångström in x-direction.

mol valencecheck

```
mol valencecheck ehandle label ?failedatomvariable? ?nitrogenmode?
m.valencecheck(?variable=?,?nitrogenmode=?)
```

Perform a valence check on the molecule, comparing the current bonding situation at all atoms to the list of element-specific valence states in the system element table. This command is intentionally quite picky, discouraging for example the use of pentavalent nitrogen by default. For the calculation of valence, only bonds of type *normal* (VB bonds) are taken into account. *Complex* bonds and pseudo bond types thus do not interfere in the calculation. Some more exotic metal atoms with many different valence states, or few well-defined covalent compounds, such as *vanadium* or *rhodium*, always pass.

The handling of nitrogen in pentavalent or ionic form can be controlled by setting the optional *nitrogenmode* argument, or modifying the global **`::cactvs(nitrogen_valence_check)`** variable.Possible values are *xionic*, *ionic* (the default), *asis*, *penta* and *xpenta*. These are the same values as with the **`ens nitrostyle`** command - please refer to that command for more information. In *asis* mode, both ionic and pentavalent forms pass.

The return value of this command is the number of atoms which failed the valence check. If the optional parameter *failedatomvariable* is specified as non-empty string, it is the name of a variable which is set to a list of the atom labels or references which did fail, or is set to an empty list in case no problems were found.

Note that this command assumes that all hydrogen atoms are in place. Checking molecules with implicit hydrogen atoms is not supported.

Example:

```
mol valencecheck [ens create {CN(=O)=O.C[N+](=O)[O-]}] 1 badatoms
mol valencecheck [ens create {CN(=O)=O.C[N+](=O)[O-]}] 2 badatoms
```

These sample commands check the valence situation of *nitromethane* in two encoding formats. The first molecule, using a pentavalent nitrogen encoding, returns 1, indicating one failed atom, and the variable *badatoms* is set to 2, the label of the pentavalent nitrogen. The second molecule, checked with the line below, passes without problems, with a return value of 0 and an empty *badatoms* variable.

The *molfile* Command

The *molfile* command is the generic command used to manipulate chemical structure and reaction files. These can be of any supported format, not just **MDL** molfiles.

Molfiles are major objects. They are uniquely identified by their label alone. *Molfiles* do not contain minor objects.

Example:

```
set fhandle [molfile open myfile.sdf]
set ehandle [molfile read $fhandle]
molfile get $fhandle record
```

As explained in more detail in the section about working with structure files, the *molfile* handle identifier can be replaced by a file name. This file is automatically opened, the command executed, and the file closed in a single one-shot operation.

In the context of structure files, file-related data is usually provided as attributes. However, *molfiles can* store property data like any other chemistry object.

Example:

```
molfile get $fhandle F_COMMENT
```

When property data is requested which is not of the molfile type, the next record is read from the file into a temporary ensemble, reaction or dataset object, depending on the file configuration. An attempt is then made to obtain the property data from that object. Afterwards, the object is automatically deleted.

Example:

```
set mw [molfile get „somefile.smi‟ E_WEIGHT]
```

This example temporarily opens the file, reads the first record into an ensemble, and computes the molecular weight. Both the ensemble and the molfile object are transient and do no longer exist after the command completes.

This is the list of currently officially supported subcommands:

molfile add

```
molfile add filehandle ?objecthandle/objecthandlelist?...
f.add(objectsequence/objectref,...)
Molfile.Add(filename,objectsequence/objectref,...)
```

If the *filehandle* argument refers to an open chemistry data file, this command is indistinguishable from **`molfile write`**.

A difference only exists if the *filehandle* argument is a file name (or, in cased of **PYTHON**, the class method is used). In that case, **`molfile write`** overwrites an existing file, while **`molfile add`** attempts to temporarily open the file for appending, as with a **`molfile open`** ***`filename`*** **`a`** command. If the format of the output file supports appending, the output objects are written as new records after the last existing record.

It is not possible to append to single-record file formats.

Example:

```
molfile add smilerecords.smi $eh
```

molfile append

```
molfile append filehandle ?property value?...
f.append({?property:value,?...})
f.append(?property,value,?...)
```

Standard data manipulation command for appending property data. It is explained in more detail in the section about setting property data. This is not a command to append file records. Use the **`molfile write`** command for this purpose.

The command returns the first data value.

Example:

```
molfile append $fh F_GAUSSIAN_JOB_PARAMS(route) "Opt=(AddRed,CalcFC)"
```

molfile assign

```
molfile assign filehandle srcproperty dstproperty
f.assign(srcproperty=,dstproperty=)
```

Assign property data to another property on the same ensemble. Both properties must be associated with the same object class. This process is more efficient than going through a pair of **`molfile get/molfile set`** commands, because in most cases no string or **TCL/PYTHON** script object representations of the property data need to be created.

Both source and destination properties may be addressed with field specifications. A data conversion path must exist between the data types of the involved properties. If any data conversion fails, the command fails. For example, it is possible to assign a string property to a numeric property - but only if all property values can be successfully converted to that numeric type. The reverse example case always succeeds, out-of-memory errors and similar global events excluded.

The original property data remains valid. The command variant **`molfile rename`** directly exchanges the property name without any data duplication or conversion, if that is possible. In any case, the original property data is no longer present after the execution of this command variant.

The command returns the object handle for **TCL**, or object reference for **PYTHON**.

molfile backspace

```
molfile backspace filehandle ?nrecords?
f.backspace(?records=?)
```

Position the file pointer backwards. If no record counter is specified, the file is backspaced by a single record. It is an error to attempt to reposition the file before the beginning of the file.

Examples:

```
molfile backspace $fh
molfile set $fh record [expr [molfile get $fh record]-1]
```

These two sample lines provide identical functionality.

The **`molfile backspace`** command is often used in combination with the **`molfile copy`** command in order to copy records with specific properties verbatim:

```
set eh [molfile read $fh]
```

```
if {[strucuture_passes_condition $eh]} {
   molfile backspace $fh
   molfile copy $fh $outfilehandle
}
```

molfile blob

```
molfile blob enshandle/reactionhandle/datasethandle ?attribute value?...
molfile blob enshandle/reactionhadle/datasethandle? ?attribute_dict?
Molfile.Blob(eref/xref/dref,?attribute,value?,...)
Molfile.Blob(eref/xref/dref,attribute_dict)
```

This is an alias of **`molfile string`**. Please refer to the section on that command for more information.

molfile close

```
molfile close ?filehandle? ...
molfile close all
f.close()
Molfile.Close("all")
Molfile.Close(mrefsequence/mref/mhandle,...)
```

Close one or more file handles. If the file handle corresponds to a scratch file, the file is deleted. If it corresponds to a pipe, all programs in the pipe are shut down.

If *all* is passed instead of a set of file handles, all currently opened structure files are closed. Standard **Tcl** or **Python** files upon which a *molfile* handle has been piggybacked are not affected, i.e. these language channels are flushed and remain open, while the *molfile* object component is closed.

It is a good idea to close files when they are no longer needed. In addition, while most file format I/O modules commit all data to disk after each record has been written, so that a clean close-down is not absolutely required, there are file formats for which the I/O module has a cleanup or finalization routine which is only called if the file is properly closed.

The command returns the number of files which were closed.

Example:

```
set fhandle [molfile open scratch]
molfile close $fhandle
```

The example closes a scratch file, which is automatically deleted from disk when it is closed.

On normal interpreter program exit, the close functions of all remaining open file handles are automatically called.

molfile copy

```
molfile copy filehandle ?channel? ?count? ?startrecord/startrecordlist?
f.copy(?outfile=?,?count=?,?startrecord=?)
```

Copy a record to a **Tcl** or **Python** file I/O channel, to a **Cactvs** structure file handle, or retrieve it as a byte image. No interpretation or formatting of the data in the file record(s) takes place - the data is copied verbatim, byte by byte.

If file format conversion is desired, the data items (ensembles, reactions, datasets) must be explicitly read (**`molfile read`** command) as chemistry objects and written to another **`molfile`** opened for output in the desired format (**`molfile write`** command). That procedure involves re-formatting and potential loss of formatting or information which was not captured by the input routine, or cannot be written by the output routine.

By default the next record after the current file pointer position is returned as a byte image. The optional parameters allow the selection of a specific start record (beginning with 1 for the first record), the copying of multiple records in one command (by default, a single record is processed), and output to alternative **Tcl** or **Python** file I/O channels or **Cactvs `molfile`** structure file handles. If an empty string, **`None`** in **Python**, or the value 0 are used as start record number, the file is copied from the current position. If the start record is negative, it is interpreted as offset from the current position. Therefore, passing -1 as parameter instructs the command to backspace by one record prior to copying. Not all files can be backspaced. The start record can also be specified as a record list (**Tcl**) or record sequence (**Python**). In that case, the input file pointer is positioned to every specified record in order, and from that position the selected number of records is copied. If the special record count values *end* or *all* are used, all remaining records in the input file are copied. Otherwise, if the number of available records is smaller than the requested copy count, an error results.

If the output channel argument is omitted, or set to an empty string, the record(s) are returned as a byte sequence command result. Otherwise, the data is written to the file handle the argument is connected to. For **Cactvs `molfile`** handles, the destination is the current write position of the underlying file handle. On **Unix**/**Linux** systems, writable active **Tcl** file or socket handles (in the form *filexxx* or *sockxxx*) are also supported, but not on Windows. Additionally, the special output channel names *stdout* and *stderr* can be used. If output is written to a channel, and not returned as blob, the number of actually copied records is returned as the command result.

The I/O modules for some formats like **SDF** provide optimized fast copy routines and are thus notably faster to copy then other file formats without explicitly encoded record positions. These still need to read the file line by line and maintain a parser state, though they can avoid decoding the record contents as structures or reactions.

Example:

```
set eh [molfile read $fhandle]
set fhout [open "metal_compounds.sdf" w]
if {[ens atoms $eh metal exists]} {
   molfile copy $fhandle $fhout 1 [expr [molfile get $fhandle record]-1]
}
```

This example reads a structure from an input file, checks whether is contains a metal atom, and if yes, copies the record unchanged to an output file, which is opened as a simple **Tcl** text file channel in this example. The expression which forms the last parameter backspaces the input file by one record, so that the same record which was just read can be copied. A simpler solution for the same functionality is to simply pass -1 as argument. This works of course only if the input file can be repositioned backwards. i.e. normal text files are fine, standard input or a socket connection do not work.

molfile count

```
molfile count filehandle ?maxrecords? ?readscope?
f.count(?maxrecords=?,?readscope=?)
Molfile.Count(filename,?maxrecords=?,?readscope=?)
```

Count the number of records in the file.

If the file format contains an internal or external record index with information about the complete file, the answer is produced from the index, and thus is typically obtained fast. Otherwise, the file is skipped from the current position until the end, and the sum of the number of records encountered while skipping and the record index when the count started is returned. In case of files which are rewindable, the original input file pointer position is then be restored. On non-rewindable files, the file contents are consumed, and no return to the old input position is possible. For files which are opened for writing, the count usually is simply the current output position, except for those few file formats which support in-file record replacement in combination with a complete file index. In the latter case, the count is again extracted from the index.

During the record skipping part the file contents are not physically read if possible. Rather, the skip function of the responsible file format I/O module is used to scan the file effectively. After arriving at the end of the file, a full in-memory record position index has been assembled for the file, and future record selection within files which support re-positioning is fast.

The type of record boundaries counted depends on the input scope of the file. For file formats which support multiple input modes, such as for extraction of ensembles or molecules or datasets, the count is dependent on the type of object which is configured to be read. If the file input object type is changed, the in-memory record index table is discarded.

If the *maxrecords* parameter is specified, and is not a negative number, it is the maximum count reported. No attempt is made to position the file beyond this mark during the count process. This has no effect on future input operations - these may still proceed beyond the reported count. This option is not intended to be generally useful, but is used for example in the structure browser *csbr* with the *-m* option to enable quick inspection of a file without full scanning.

The optional *readscope* parameter can be used to temporarily modify the read scope under which the file is processed. It can be any of the generally recognized values (*mol, ens, reaction, dataset*). If the file format does not support the specified mode, its default mode is silently used. If the file is not positioned at the beginning of the data, the count reports the sum of the currently known records as perceived by the previous read scope, and the remaining file records under the new one. If these values are different, the result may only be useful under very specific circumstances. The the parameter is not set, or an empty string is passed, the currently set, or, for one-shot file operations, the default read scope, is used.

Example:

```
set nrecs [molfile count "thefile.sdf"]
set nrecs [molfile count "test.spl" -1 mol]
```

molfile dataset

```
molfile dataset filehandle ?filterlist?
f.dataset(?filters=?)
```

Return the handle of the dataset associated with the file handle. If no such dataset is set, the command returns an empty string, or **None** for **PYTHON**. The command

```
molfile get $filehandle dataset
```

is equivalent.

This command is different from the dataset commands for ensembles, reactions or tables, where it indicates membership in a dataset. File objects cannot be a member of a dataset. This dataset association is explained in more detail in the **molfile set** command section.

molfile defined

```
molfile defined filehandle property
f.defined(property)
```

This command checks whether a property is defined for the structure file. This is explained in more detail in the section about property validity checking. Note that this is *not* a check for the presence of property data! The **`molfile valid`** command is used for this purpose.

molfile delete

```
molfile delete filehandle recordlist ?rebuildindex?
f.delete(records=,?rebuildindex=?)
Molfile.Delete(filename,records=,?rebuildindex=?)
```

Delete records from the file. The file must have been opened for writing or update, and be rewindable. In case the file is not a simple record sequence, the I/O module for its format must provide a deletion function, or the operation will fail.

The deletion record list is a single or set of record numbers in any order. They are sorted and duplicates removed before file modification commences. It is no error to specify an empty removal record list. The record numbering starts with one, and the record numbers are referring to the record numbering at the moment the command is issued. There is no need to compensate for intermediate record numbering shifts when more than one record is deleted.

The optional index rebuild parameter, a boolean value, can be set to optimize the deletion process for files in formats which maintain field index information. By default, indices are updated as part of the deletion process. In case many records are deleted, it may be more efficient to drop the indices prior to the deletions and rebuild them after the records have been removed. In order to select this alternative procedure, a *true* parameter value can be set. At this time, the only file format which actually can use that parameter is the *bdb* database file format.

In case the file is to be truncated, the **`molfile truncate`** command is usually more efficient.

This command returns the number of deleted records. It does not close or destroy the file handle, or the underlying file.

molfile dget

```
molfile dget filehandle propertylist ?filterset? ?parameterdict?
f.dget(property=,?filters=?,?parameters=?)
Molfile.Dget(filename,property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **`molfile get`** command. The difference between **`molfile get`** and **`molfile dget`** is that the latter does not attempt computation of property data, but rather initializes the property values to the default and return that default if the data is not yet available. For data already present, **`molfile get`** and **`molfile dget`** are equivalent.

The **Python** class method is a one-shot command. The transient *molfile* created from the initialization items is automatically closed when the command finishes.

molfile dup

```
molfile dup filehandle
f.dup()
```

This command duplicates a file handle. The duplicate handle or reference points to the same underlying file or other data channel, is opened in the same access mode, and positioned at the same record. Also, all file object attributes and file properties are set to identical values.

Currently, it is not possible to duplicate virtual file sets opened by a **`molfile lopen`** command.

The command returns a new file handle or reference.

molfile exists

```
molfile exists filehandle ?filterlist?
f.exists(?filters=?)
Molfile.Exists(mref,?filters=?)
```

Check whether a molfile handle is valid. The command returns 0 or 1. Optionally, the molfile may be filtered by a standard filter list, and if it does not pass the filter, it is reported as not valid.

molfile extract

```
molfile extract filehandle retrievallist
f.extract(retrievallist)
```

Extract the contents of data fields from the file, without reading full structure or reaction records if possible. This operation requires a support function in the I/O module for the file format. Generally, only formats optimized for query operations, such as the **CACTVS** *bdb* and *cbs* formats provide such a function in their I/O module.

This command is essentially a shortcut for a **`molfile scan`** command with an empty query condition and a *propertylist* retrieval mode. Please refer to that command for details about the possible contents of the retrieval list.

The result is a nested list of extracted property values, with one outer list element for every file record to the end of the file, and inner list with one element per retrieval field.

molfile filter

```
molfile filter filehandle filterlist
f.filter(filters)
```

Check whether the structure file passes a filter list. The return value is boolean 1 for success and 0 for failure.

Example:

```
molfile filter $fhandle $filter
```

molfile fullscan

```
molfile fullscan filehandle queryexpression ?mode? ?parameterdict?
f.fullscan(query=,?mode=?,?parameters=?)
Molfile.Fullscan(filename,query=,?mode=?,?parameters=?)
```

This command is the same as **`molfile scan`**, except that an automatic rewind (see **`molfile rewind`**) is performed before the query is executed. The same effect can be achieved by setting the *startposition* parameter value to 1.

molfile get

```
molfile get filehandle propertylist ?filterset? ?parameterdict?
molfile get filehandle attribute
f.get(property=,?filters=?,?parameters=?)
f.get(attribute)
f[property/attribute]
f.property/attribute
Molfile.Get(filename,property=,?filters=?,?parameters=?)
Molfile.Get(filename,attribute)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

The molfile object possesses a rather extensive set of built-in attributes, which can be retrieved with the **`get`** command (but not its related subcommands like **`dget, sqlget`**, etc.). Most of them can also be manipulated with a **`set`** command. In addition, *molfile* objects can possess file-level properties. The standard prefix for these is `F_`.

The **PYTHON** class method is a one-shot command. The transient *molfile* created from the initialization items is automatically closed when the command finishes.

Example:

```
set c [molfile get $fhandle F_COMMENT]
```

These built-in attributes are:

- *address_city*
  The city part of the author contact address.
- *address_country*
  The country part of the author contact address, following the ISO3166 standard.
- *address_state*
  The state part of the author contact address. Empty if not applicable.
- *address_street*
  The street address part of the author contact address. Includes floor, house number, etc.
- *address_zip*
  The **ZIP** code or other applicable postal code of the author contact address.
- *affiliation*
  The institution the author works for.
- *affiliationduns*
  The **DUNS** registration ID of the affiliated institution. This is primarily useful for US government projects.
- *affiliationurl*
  The **URL** of the affiliated institution.

- *atomlabelproperty*
  The name of a property which holds data for a parallel user-defined atom numbering scheme (see *writeflags*/*writelabels* attribute) which can be output by some I/O modules. The default property is `A_LABEL`. The property must be associated with atoms, but is not required to be an integer, if the I/O modules supports alternative data types (i.e. for **CDX**/**CDXML** the label data in the file format is internally a string, and any different property data type is converted as necessary). This attribute has an effect only if the *writelabels* flag is also set in the *writeflags* attribute.
- *author*
  The author of the file, as free-form string data.
- *authorization*
  A service authorization **URL**, which might for example be presented to the user for approval of access to a resource. In the case of *dropbox* file access, this data is copied from the global value of the I/O module (see **`filex get`** command). For normal files, this attribute is empty, and setting it to a string value has no effect.
- *authorurl*
  A **URL** with information on the author, or an empty string if unset.
- *batchsize*
  The number of records in a standard processing batch. The default batch size are 10 records.
- *bondlength*
  The standard bond length to be used in the file. The unit is points (1/72 inch). If the value is negative (the default), the standard format-specific bond length is used. This attribute is only supported in a few graphics-oriented file formats, such as **CDX** or **SKC** files, or **EMF** images.
- *cachesize*
  The size of the record prefetch cache the file should use. Normally, the size is zero and no such cache is employed. The I/O modules for a few file formats, such as **PUBCHEM** CID and SID files, where the individual retrieval of a record via the Internet is almost as expensive as fetching a sizable batch, use a cache if allowed and prefetch multiple records when a record read operation is performed and the cache is empty or the requested record is not in the cache. A later read can, if the input record is in the cached set, return the data without establishing a new network connection.Using a cache is beneficial only when the expected access pattern is linear and in ascending record order. It decreases performance if the record access pattern is random and not limited to a continuous record set that fits into the cache.
- *category*
  A category string to be used if the file is stored in a repository.
- *chain*
  A single-letter code indicating the chain to be read from records with structure disorder data. These can for example be found in **PDB** files. The default value '?' automatically selects the first chain which is encountered in the file record. After a record has been read, the attribute is set to the actual character of the chain which was selected, so it needs to be reset in case more than one record is input via this file handle. If the chain character is set to an empty string, all atoms are read from files even if they belong to multiple overlapping disordered structure instances. This can of course lead to problems in connectivity representation. The alternative name *disordered* is an alias for this attribute.

- *classuuid*
  The base class **UUID** of this file object, copied from its I/O module **UUID**.
- *coords*
  If the toolkit was compiled with factory support, these are the coordinates of the object icon on its workbench, encoded as integer pair. This attribute can be changed
- *compact*
  A boolean flag indicating whether the file is present in a abridged form, or should be written as compact as possible. This attribute effects few file formats. An example is the native **Cactvs** ASCII format (*cascii*).
- *complexresolver*
  A boolean flag which enables or disables bond type processing after input. It the flag is on, typical complex bonds between metal atoms and ligands, or between metal atoms, are recognized and re-coded as *complex* bonds, which provide connectivity, but do not participate in valence electron counting. In many cases, this improves the general representation quality of the structures. However, since most chemical data exchange formats do not support this type of bonds, it can also make export of the data difficult. By default, this flag is *on*. For maximum portability, it should be switched off. This attribute is a convenience shortcut operating on the *readflags* attribute.
- *computationlog*
  A read-only attribute. It is a list of all properties which were computed during a record write operation, either implicitly or explicitly via the *computelist* attribute. This can be used to determine which effects the output has had on the information content of a written object, or to optimize I/O throughput by performing pre-computation of these properties in a separate thread.
- *computelist*
  A list of properties which are automatically computed before an output object is written if they are not yet valid. Computation failures are silently ignored.
- *compression*
  The detected file compression type. It can be one of *none*, *compress*, *pack*, *gzip* or *bzip2*. Compressed files are automatically opened for reading via a pipe to the suitable decompressor program, if it can be located. This attribute can also be set, but it currently has no effect on the actual output in any format. In order to write compressed files, open an output as a pipe to a compressor program.
- *corsdomain*
  If specified, and the *httpheader* attribute is also set to a value larger than zero, the output **HTTP** header contains a **CORS** domain header line (*Access-Control-Allow-Origin:*). Useful values for this parameter are either * (for free access), a host name, or a domain name. If this parameter is not specified, **HTTP** headers do not contain information, which usually is equivalent to allowing access for **AJAX** queries only from the same server as the requesting page.
- *ctime*
  A read-only attribute reporting the time of the last status change of the file. Its unit are seconds since January 1st, 1970. This value is meaningful only for normal disk files.
- *date*
  The date the file structure was defined. This is not the same as the filesystem time stamps, such as the *ctime* or *mtime* attributes.

- *deletable*
  Flag indicating whether this *molfile* can be deleted or closed with a standard **`molfile close`** command. The attribute is read-only. *Molfiles* which are, for example, property data values or a part of a **`molfile loop`** command cannot be deleted by standard means.
- *deselection*
  This somewhat awkwardly named attribute is the inverse of the selection attribute. For further explanation, refer to the paragraph on selection.
- *device*
  A read-only attribute reporting the device number of the file. This is meaningful only for normal disk files, and only supported on Unix/Linux.
- *doi*
  A digital object identifier for the file content, if defined.
- *downloadfilename*
  If the *httpheader* attribute is set to a value larger than zero, this value is included in the **HTTP** header as save file name, with a **MIME** type of application/save-to-disk which overrides any native file format **MIME** type.
- *droplist*
  A list of properties which are not to be written to the file, even if they are already present on output objects and the file format can encode them. Naming a property in this list does not delete them from the property set of objects which are written to the file, just suppresses their output.
- *email*
  A contact email of the file author.
- *embedformat*
  The format of embedded objects encoded in another format. This is meaningful only for a few file formats, for example *zip* (which contains single-record files of a different type) or *rtf* (which may contain *cdx* or *skc* embedded **OLE** objects). If this attribute is not set, the default depends on the wrapper format (i.e. *SDF* files for *zip*, *cdx* **OLE** objects for *rtf*). Setting it to an empty string or *none* disables embedding where applicable. The attribute is updated on input and can be read when a file record is input which contains embedded data.
- *encoding*
  The detected encoding type of the file. It can be one of *ascii*, *binary* or *unicode*. This is a read-only attribute.
- *eof*
  This read-only boolean attribute indicates whether the file read pointer is at the end of the file.
- *eolchars*
  A sequence of characters which are used as line terminators for the output of text-based file formats which do not define a specific line end character. The default value is platform-dependent. It is a single newline character on Linux/Unix, **`CR/LF`** on Windows and a single **`CR`** on Macs. This attribute has no effect on input. All input routines automatically recognize and read all three variants on all platforms.

On setting, the magic strings *windows*, *mac* (both checked for the first three characters only) as well as *unix* and *linux* are translated to the standard platform line terminators and not copied verbatim. Alternative names for these standard system encodings are *crlf*, *cr* and *lf*. The special value *default* resets the attribute to the platform-dependent default.

- *eor*
  A read-only attribute which indicates at what type of record terminator the current read position is located. Possible values are *none*, *mol*, *ens*, *reaction* and *dataset*. The *none* value indicates that reading did stop in the middle of a record due to some problem.
- *errorproperty*
  A read-only attribute which holds the name of the last property where input failed. This is not supported by all file I/O modules. It is especially useful for binary formats where a line number cannot be used for simple visual inspection of an input problem.
- *failures*
  A list of properties for which computation failed on this object. This is a read-only attribute. Depending on configuration settings, this information may be used to block pointless attempts at re-computation of incomputable data.
- *fd*
  A read-only attribute which reports the system channel number the file object is associated with.
- *fields*
  This is a list of the names and potentially attributes of data fields in the file. For simple formats such as SD files, this is simply a list of property names, and it is updated after each read record to track a potentially changing field set. For more complex formats such as *bdb* and *cbs*, every list item is a nested list which contains the field name, field flags, field object class association and partition file. The field output for simple formats such as SD is controlled via the *writelist* attribute, and the value of the fields attribute has no effect on output. However, the I/O modules for complex database-type formats such as *bdb* and *cbs* provide a handler function which translates an updated value of this attribute into a changed database layout. Depending on the I/O module, this may be supported only for an empty file (*cbs*), or may be possible even for files which already store records (*bdb*). This attribute can also be addressed by the alias *fieldnames*.
- *filelock*
  On reading, this is a boolean flag indicating whether a file lock is currently set on the file or not. On setting, the argument can be *release*, *trylock*, *forcelock* or *test*. The first variant attempts to release an existing file lock, the second variant attempts to set a file lock, but returns immediately if that is not possible, the third variant blocks until the lock can be established, and the fourth version tests for the presence of a lock. The return value is a boolean status result. This command is not supported on Windows. File locking may pose special problems if the file is not residing on a local file system.The underlying system call is *lockf64()* or *lockf()*. Please consult your operating system manual for more details.
- *fileset*
  A read-only attribute containing a list of the names of the physical files which are behind the file handle. For normal files, this is a single list element for a single file. However, for file handles opened by means of the `molfile lopen` command to access a virtual file assembled from multiple physical files, this can be a list with more than one element.

- *filter*
  A query expression (see **`molfile scan`** command) which input records must match to yield a result object when a **`molfile read`** command is run. The read command is automatically looped until a matching record is found, or the end of the input source is reached. Since the test is only applied after a prospective input object has already been fully read internally, this style of record filtering is in many cases considerably less effective than using **`molfile scan`** for file formats which possess query acceleration features, such as **CBS**, **BDB** or the **PUBCHEM** virtual file module. For the reading of simple text files, such as **SDF**, there is no performance difference to using **`molfile scan`** in the ens or reaction object retrieval mode, and this type of filter which can be easily adjusted or disabled (by setting it to an empty string) can be convenient.
- *fontsize*
  The standard font size for text in graphics-oriented formats, such as **CDX** or **SKC**. The value is a floating point number measured in points (1/72 inch). A value of zero or less, which corresponds to the default, lets the software chose a suitable value, which is dependent on scaling and bond length.
- *fold*
  The number of characters after which the software should look for a good position to use a continuation character and line break. This is only used in a few formats, such as **SLN**.
- *footer*
  If the toolkit was compiled with factory support, this is the footer of the object icon on a workbench. This attribute can be changed.
- *format*
  The standard name of the file format the *molfile* object is linked to. This is normally only set in scripts for output files, because the format for input files is auto-detected. Nevertheless, it is possible to set a format explicitly also for input files, and even to switch it when records have already been read. When setting a format, generally a set of alias names are recognized in addition to the short official name.
- *from*
  The sender of a file. This is only set when the file has been extracted from a mail message or attachment.
- *gflags*
  If the toolkit was compiled with factory support, this is the currently set object icon rendering flag collection.
- *handle*
  The handle of the file as a read-only attribute. Not generally useful, because in standard access modes you already need the handle to identify the file object.
- *header*
  If the toolkit was compiled with factory support, this is the header of the object icon on a workbench. This attribute can be changed.
- *height*
  The maximum height of a structure or reaction depiction in points (1/72 of an inch). This is only used for graphics-oriented formats, such as **CDX**, **SKC** or **EMF**. If the attribute is set to a negative value, which is the default, the size is indirectly controlled by the bond length and atom coordinates. In case this attribute is set to a positive value, and the depiction would exceed the maximum height, it is automatically scaled down proportionately.

- *hidden*
  Flag indicating whether the object is hidden. This is not the same as the *invisible* state. This attribute is intended to be used for rendering object selections. This attribute can be set.
- *highmaprecord*
  The maximum record to include in a memory-mapped section of the file for accelerated read access. If set to a negative value, which is the default, the system automatically determines if mapping is worthwhile, and if it is, map the full file. This attribute is primarily useful for the acceleration of queries which repeatedly operate in a section of a larger file, for example when running distributed queries with multiple processes handling different parts of a large file.
- *host*
  This is a shortcut for the host name part of a file or virtual file addressed via an URL. For simple retrieval it is equivalent to the URL field attribute *url(hostname)*. For some I/O modules, for example the interface to access **MYSQL** tables as virtual structure files, a change of the host name does have an effect and results in (re)-connection to a different database host. For normal files accessed via a URL a change of the attribute is ignored after the file has been opened. Files that are not associated with an URL have an empty host name value.
- *httpheader*
  Configure whether the file content output should be prefixed by a **HTTP** header. This can be useful when scripting **CGI** or **FCGI** applications. The value can be 0 (no **HTTP** header prefix, the default), 1 (standard **HTTP** header prefix, without status code) or 2 (**HTTP** header including 200 status code). String values *none*, *default* and *status* are also recognized. Not all table I/O modules support this feature. This attribute has an effect only if the file is opened for output, and has no records when it is opened. The data is written immediately after the file is opened, before any other output is performed. The configured *corsdomain*, *mimetype* and *downloadfilename* attribute data is part of the output. Note that prefixed files are only useful in Web application **CGI/FCGI** contexts, where the header is stripped before the data is seen. They cannot be read as normal structure files.
- *hydrogenfilter*
  A hint about the desired output style of hydrogen atoms of the structure. In contrast to the *hydrogens* attribute, this hint does not actually change the structure by adding or removing hydrogen atoms, neither on the original output object nor a temporary processed structure or reaction duplicate. Not all I/O modules support this flag. Its availability can be queried via the *capabilities* attribute of the `filex` command for the format. The possible values are *default* (or -1), which is the default and selects the default hydrogen write mode of the file format, *none* (or 0) which suppresses hydrogen output, *special* (or 1) which writes hydrogens shown normally with a symbol only, and *all* (or 2), which writes all extant hydrogens. Since this attribute does not change the hydrogen atom set, setting for example the mode to all when there are no hydrogens attached to the structure has no effect.
- *hydrogens*
  The hydrogen processing mode of the file. Its default can be controlled via the system variable *::cactvs(default_hydrogen_addition_mode)*. Its standard setting is *asis*, meaning the hydrogen set is to kept as it stored in the objects for output, or defined in the original file records for input. Possible modes for this attribute, or the system control variable, are *add* (add a complete standard set of hydrogens), *asis* (keep unchanged), *strip* (strip hydrogens except those which are normally displayed, such as bonded to hetero atoms or at stereo centers), *stripall* (strip all hydrogen), *stripadded* (strip all hydrogens which were added by

a **hydrogen add** command, automatic hydrogen addition on input, or similar mechanisms) and *addblind* (which is the same as add, but does not register the added hydrogen atoms as implicit in property A_IMPLICIT). When writing a structure object to a file with enabled hydrogen processing, the original object is not changed. Hydrogen processing takes place on a ephemeral duplicate object. On input, hydrogens which are no explicitly encoded, but defined via implicit valence rules in the format specification are still instantiated in *asis* mode. For example, a single C atom in an **MDL** Molfile is read as a single atom, because there are no default valence rules, but a C as a **SMILES** string is expanded into one carbon plus four hydrogen atoms. For a method to suppress the expansion of valence-implicit hydrogen atoms, see the *readflags* attribute.

- *hydrogenstatus*
  An enumerated value providing information about the hydrogen status of the file. Possible values are *unknown*, *complete* (all hydrogens present), *partial* (some hydrogens present) and *missing* (no hydrogens present). This attribute is updated when data is read from files which encode this information. It may also be set and has an effect on some post-processing operations on objects read from the file.
- *ignoreempty*
  A boolean flag which instructs, when set, the I/O module of the file format associated with the *molfile* object to ignore empty records without atoms when reading from the file. By default, this flag is not set and empty records are retrieved as empty ensembles or other objects.
- *ignoreerrors*
  A boolean flag which tells the I/O module of the file format associated with the *molfile* object to ignore errors and to attempt to read or write the next record instead. By default the flag is not set and errors in I/O result in Tcl script command errors.
- *ignorelist*
  A list of properties which should not be read from the file, even if they are explicitly encoded in the records.
- *incomplete*
  This is a boolean read-only boolean flag which indicates that a record was only read partially. This is the same as checking for the presence of the *incomplete* flag in the *flags* attribute.
- *infourl*
  A **URL** with information on the file content, or an empty string if unset.
- *inode*
  A read-only attribute reporting the inode number of the file. This is meaningful only for normal disk files, and only supported on Unix/Linux.
- *instanceuuid*
  The **UUID** for this object instance.
- *invisible*
  Flag indicating whether the *molfile* object is invisible. This is not the same as the *hidden* state. An *invisible* object is no longer accessible via its handle. This is usually the case for objects which are scheduled for deletion, but still have lingering referring pointers. This attribute is read-only.

- *iscompressed*
  A boolean read-only attribute which is set when the file is compressed by one of the recognized compression algorithms (*gzip*, *bzip2* by default). In that case, the file is not accessed directly but via a pipe the the appropriate decompression program, which changes the file handling characteristics.
- *ismapped*
  A boolean read-only attribute which is set when the file is read via a memory-mapping method.
- *ispipe*
  A boolean read-only attribute which is set when the file is accessed via a pipe, either because it was explicitly opened to a pipe, or because decompression (*gzip*, *bzip2*) or character encoding (*iconv*) programs where automatically spliced in.
- *javaobject*
  If the toolkit was compiled with **JNI** support, this attribute reports the memory address of the **JNI** wrapper class instance, if it exists.
- *jstreversal*
  A boolean flag indicating whether the **JST** special encoding variant for **MDL** Molfiles should be used.
- *keywords*
  A list of keywords associated with the file.
- *lastrecord*
  The value of the file record read position before the last **`molfile read`** command. This is normally the value of the *record* molfile attribute after the read operation minus one and corresponds to the file record number of the read object in the data file.
- *license*
  The license class associated with this file. Setting the license to a standard type updates the associated **URL** with a standard location.
- *licenseurl*
  A **URL** with details about the network object license.
- *literature*
  A free-form literature reference.
- *line*
  A read-only attribute returning the current line number. *lc* is an alias name for this attribute. Generally this attribute is meaningful only for text-based file formats. For most binary formats, the value of this attribute is the same as the record number. This line number always refers to the current physical file. To get the global line number of a virtual file set, use the *vline* attribute.
- *loinc*
  The **LOINC** code (**`https://loinc.org/`**) describing the file contents. For I/O formats where this information us used (currently **SPL**) and the attribute is not set, the default is **`64124-1`** (*Indexing - substance*). Setting a **LOINC** value for the first time takes a second or two because this is a controlled vocabulary, and the term table is loaded from disk for verification.
- *loopitem*
  The current file input item in a **`molfile loop`** statement. This is the same as the content of the loop variable. If no loop is active, this is an empty string. This is a read-only attribute.

- *lowmaprecord*
  The minimum record to include in a memory-mapped section of the file for accelerated read access. If set to a negative value, which is the default, the toolkit automatically determines if mapping is worthwhile, and if it is, map the full file. This attribute is primarily useful for the acceleration of queries which repeatedly operate in a section of a larger file, for example when running distributed queries with multiple processes handling different parts of a large file.
- *mailencoding*
  This is a read-only attribute which is only set if the file has been extracted from an email message or attachment. Possible values are *unknown*, *ascii*, *iso* (for ISO 8859-1), *quoted* (for quoted printable), *base64* and *utf8*.
- *mailproperties*
  This is a read-only attribute which is only set if the file has been extracted from an email message or attachment. It is a list of properties which were requested for computation in a header field. This attribute is typically used for setting up email-based property computation services.
- *maxblobsize*
  The maximum size of **CACTVS** ensemble or reaction blobs which are part of the file records, measured in bytes. This attribute only applies to those few file formats which store structure and reaction data as **CACTVS** toolkit blobs. Currently these are **CBS** and **BDB**. If the blob size exceeds the limit, the input or output of the record fails. The default value are 256K, which is more than sufficient for standard applications. If the attribute is changed, a minimum value of 64K is silently enforced. Increasing the attribute can have a small negative effect on I/O performance, but is otherwise safe.
- *mimeboundary*
  This is a read-only attribute which is only set if the file has been extracted from an email message or attachment. This is the string which was used to separate **MIME** data blocks in the message.
- *mimedefaulttype*
  A read-only attribute giving the default **MIME** type associated with the current file format.
- *mimetype*
  The currently configured **MIME** type for the file. Initially, it is set to the default type (attribute *mimedefaulttype*). However, it can be changed, and it is used for transmitting the file data via various types of Internet connections.
- *modcount*
  The *molfile* object modification count. This is a read-only attribute.
- *mode*
  This is a read-only attribute which describes the general file access mode which was established when the file handle was created by a `molfile open` or `molfile lopen` command. Possible values are *append*, *pipe*, *read*, *string*, *write* and *update.* Note that in this attribute there is no difference between the standard read and the restricted read-only modes (see `molfile open`). The file mode cannot be changed at a later time by directly changing the *mode* attribute. However, with some limitations, a file may be switched back and forth between input and output modes with the aid of the `molfile toggle` command.

- *mtime*
  A read-only attribute reporting the time of the last modification of the file. The unit is seconds since January 1st, 1970. This value is meaningful only for normal disk files.
- *mutexcount*
  The number of recursive mutex locks held for this object. Only supported on Linux.
- *name*
  On input, this attribute simply reports the full path name of the underlying file, or the original magic name in case of special files. This attribute can also be set, and in case of normal disk files, the physical file is renamed, too, if the file access permissions are sufficient for this operation.
- *nitrostyle*
  The nitro (and similar) group encoding conventions associated with the file handle. There are actually independent settings of this attribute for input and output. The version reported by the command is dependent on whether the file is in input or output mode. Possible values are *asis*, *ionic neutral, xionic* and *xneutral*. The default input value is *ionic*, while the default output value is *asis*. When the value is modified, the new value is stored both for input and output. If the value is not *asis* and a structure item is read, its nitro group (and related groups) connectivity is automatically adjusted to the preferred style. If processing is requested for output, the connectivity change is performed on a temporary duplicate, so that the original output object is not modified.
- *nullstring*
  A string which on input is used to identify `NULL` values, or used on output to encode `NULL` values. This attribute is only used by a few I/O modules. The most important application is in reading text-based tables with embedded structure notations by means of the table structure I/O module.
- *offset*
  A read-only attribute reporting the current byte offset position of the read or write pointer. It is not meaningful for all types of data channels.
- *orcid*
  The **ORCID** code of the author (see www.orcid.org).
- *orientation*
  This value can be *none* (the default), *landscape* or *portrait*. It describes the orientation of a drawing area specified via the paper attribute. Few I/O modules use this information. The most important formats which implement this is are **CDX** and **CDXML**.
- *originalname*
  The name as originally used to create the *molfile* object. The standardized name, with path information in case of disk files, can be accessed via the *name* attribute. Changing this attribute has no effect on the file system. This is different from the handling of the *name* attribute.
- *pagecount*
  The number of (vertically stacked) pages in the document. This attribute is currently only used for the **CDX** and **CDXML** formats.

- *paper*
  An attribute describing the size of the drawing areas for formats such as **CDX** or **CDXML**, which can encode this type of information. Possible values are *none* (the default), *a3*, *a4*, *a5, a6, a7, b3, b4, b5, b6, letter, legal* and *executive*. The associated orientation of the drawing orientation can be set via the *orientation* attribute.
- *parameters*
  A free-form string which can be used to pass additional, non-standardized parameters to a file format I/O module. Few I/O modules use this, one example is the **XFIG** output code.
- *password*
  A file access password. It is used in various contexts, for example for authentication when using URL-based access to files, to enable the I/O of encrypted records in files which support partial data encryption, such as the **CACTVS CBS** and **BDB** formats, or to proceed with the execution of a remote query received via a listener port. In most cases a change of the attribute value after a file has been opened has no effect. An exception are modules which access database tables as virtual structure files. These will react to a changed user name with re-authentication to the database and table, which may result in different access permissions.
- *path*
  The repository path for displaying hierarchical repository trees. This attribute is independent of any file system paths.
- *phone*
  A contact phone number of the file author.
- *polysymbol*
  A free-form string used to override the standard symbol used by a file format I/O module to indicate polymer components. If set to an empty string, the standard symbol is used, which depends on the file format. The default is an empty string.
- *port*
  The number of a port on which the file handler should accept remote query requests. If set to a negative value (the default), no such requests are accepted, and in case a monitor thread was executing before the value was changed, it is shut down. If a positive port number is set, a monitor thread is automatically started as listener on the specified port.
- *position*
  This read-only attribute describes the relative position of the read or write pointer in the full file, as an integer in the range between 0 and 100. It is primarily intended to be used in progress meters and similar widgets. In case the relative position is unknown, for example because the total size of the input file is unknown, the value is zero.
- *preservelist*
  A list of properties which should not be changed if a file record is updated, even if the value in principle depends on, for example, changed connectivity of the main structure record. Currently, the only I/O module which supports this feature is **BDB**.
- *previousrecord*
  This read-only attribute is a convenience function to obtain the value of the record number of the file handle that before the current record was read. Usually, it is the same as the `record` attribute minus one, but if reactions from files where reagents and products are separate sub-records, or complete datasets were read, the difference may be larger.

- *progress*
  A user-defined progress value intended to track the state of lengthy operations on the file. It is an integer between zero and one hundred and is initially set to zero. When the argument is set, it accepts a floating point value, but the stored value is automatically rounded to the next integer and forced into the 0..100 range.
- *pyobject*
  If the toolkit was compiled with Python support, this attribute reports the memory address of the Python wrapper class instance, if it exists. This attribute is read-only.
- *pyrefcount*
  If the toolkit was compiled with Python support, this attribute contains the reference count of the Python wrapper class instance, if it exists. This attribute is read-only.
- *reactioncolumn*
  This attribute is the numerical index of a column in table-style data files which are, for example, read by the reaction table I/O module. The column is expected to contain a string notation for the reaction object which is returned by a **`molfile read`** operation. To this decoded object the contents of the other columns is attached as property data. Typically the content of the structure column is a Reaction **SMILES** string or similar line notation. A negative value of this attribute indicates that the presence of structure data in a specific column is unconfirmed. In that case, an attempt is made to determine the reaction column automatically, and the attribute is updated accordingly. However, setting it explicitly may still be required in case there are multiple columns with reaction data, or there are too many unreadable or `NULL` row entries to allow automatic determination.
- *reactionscreen*
  The name of the property which is used for bitvector screening in filtering records for reaction transform matching. Its default default value is controlled by the global variable **`::cactvs(default_reaction_screen_property)`** and is usually `X_SCREEN`. If a file is opened that contains information about the screen property set when the file was written (for example, **CBS** and **BDB** formats), this attribute is automatically set to the value stored in the file.
- *readflags*
  This attribute controls a set of input processing flags. If the attribute is queried, the result is a list of the names of all flags which are currently set. For modification, the preferred method is to use the bit manipulation prefixes for generic bitset operations. In case just additional flags should be activated, the **`molfile append`** command can also be conveniently used. There are also a few shortcut alias attribute names which set or reset selected, frequently used flags directly (*complexresolver*). The following flag names are currently recognized:

  - *none*
    no flags.
  - *aroresolver*
    resolve aromatic bonds into a Kekulé form. A frequent application is the input of records from **MDL SD** files which are not used as query structures, but where aromatic bonds in the original data are nevertheless and illegally encoded as the aromatic structure query bond type with magic bond order 4. This flag should not be used on formats which are guaranteed to already encode a proper Kekulé form. This includes all native toolkit formats (*cbin*, *cbs*, *bdb, cda, cpc*).

- *autowrap*
  When the end of the file has been reached, automatically start reading from the beginning of the file again, until the full file has been scanned once. This operation effects only the `molfile scan` command and is used there in order to perform full-file queries starting from an arbitrary position in the middle of the file.
- *basiconly*
  only read basic property data set, not full record. This is supported in **CBIN**, **CBS** and **BDB** formats in order to accelerate fast filter and query operations.
- *chargebalancer*
  Try to neutralize and balance charges.
- *chargecombiner*
  Try to merge opposing charge pairs where possible, changing the bond orders of paths between them if necessary.
- *complexresolver*
  perform a bond analysis and re-code typical bonds in metal complexes as non-VB bonds, which do not participate in valence electron counting. For a more detailed explanation, see the alias shortcut *complexresolver*. This flag is set by default.
- *continueafterhetatm*
  For **PDB** files, consider any atom line after the first `HETATM` to be a heterogen, regardless of the line type. This feature helps to cope with ligands which contain amino acid substructures and which some other **PDB** write software misclassified as part of the protein.
- *fixdoublespace*
  If set, this flag instructs I/O modules with support for this feature to read structure files which contain one spurious empty line after each data line, which unfortunately appears to happen sometimes when **DOS**-encoded files are transferred to Apple systems. This is not the same as reading **CR/LF** files on **CR**-only or **NL**-only platforms, or vice versa, which is always possible and fully automatic. This flag addresses the problem that, due to mishandling by obscure transfer software, duplicated **EOL**-markers are introduced in the file (two identical **CR/LF**, or **CR**, or **NL** pairs after each data line).
- *fixstereo*
  Remove spurious stereo descriptors on atoms and bonds which are not stereogenic.
- *fixwedges*
  Re-code wedges which are attached with the broad base to a stereo center (for example as written by IDBS software) into standard IUPAC format with tips at the stereo centers.
- *hetatmonly*
  In **PDB** files, read only `HETATM` lines.
- *ignorecr*
  Allow an isolated carriage return (**ASCII** 13) character without following **NL** (**ASCII** 10) character as data content instead of examining it as potential line break symbol. This flag is necessarily ignored on Mac-style input files which only use **CR** as **EOL** markers.

- *ignoreitherdb*
  If set, ignore any *either* attribute data for double bonds in **MDL** *Molfiles*. Instead, determine their stereochemistry from coordinates.

- *ignoreempty*
  When reading an empty record, with no atoms, from a multi-record file, ignore the record and immediately proceed with the next.

- *ignoreerrors*
  When reading a corrupted record from a multi-record file, ignore the error and instead attempt to re-synchronize and read the next record.

- *ignorenorecall*
  If set, the *norecall* field flag supported in some file formats (**CBS**, **BDB**) is ignored. By default, data from fields which carry this flag is not merged into the property set of ensembles or reactions when they are retrieved as objects from these files, as an optimization to avoid recalling data which is useful for queries, but not so much as object data (for example, screen bits, element counts). With a set attribute flag, all fields of the record are attached as property data to recalled objects.

- *ignorevisibility*
  Ignore any display attributes in the input data which would make atoms or bonds invisible in renderings.

- *latehprocessing*
  If this flag is set, the standard hydrogen addition/removal operations are performed after other selected processing steps have been performed. By default, hydrogen processing takes place before charge equilibration, radical charging, etc. This flag should be set if the hydrogen set in the file records is known to be complete, but the charge and radical situation is dubious.

- *lockmemory*
  Lock the shared memory mapping arena of the file into memory, preventing it from being swapped out. This is only supported on Linux, and has an effect only if the *sharedmap* flag has been set. Depending on the size of the arena, and the system configuration, this operation may require enhanced privileges.

- *logqueries*
  If the file formats supports operation logging, activate the log.

- *keepcoords*
  In case multiple molecules or ensembles are read in one operation, the system normally verifies that they do not have overlapping 2D display coordinates, and moves them apart if necessary. If this flag is set, the 2D display coordinates in property `A_XY` are always passed unchanged.

- *mergedata*
  In case there are repeat instances of the same data item in an input record, attempt to append it in a suitable fashion to the first property instance on the input object. By default, multiple data items with the same name are not merged, but result in multiple property data instances. This is a problem which is encountered typically while reading data from formats with limited syntactic expressiveness that cannot properly distinguish between these cases.

- *multibondcheck*
  attempt to correct unlikely clusters of multiple bonds.

- *nocoordinatecheck*
  do not attempt to discover and fix mixed-in missing 2D or 3D coordinates, for example encoded as all-0 values. All coordinate data is to be preserved verbatim.

- *noorigin*
  do not register the origin of the property data values from the current file as metadata information.

- *noeof*
  do not attempt to detect EOF. More data may be coming.

- *noimplicith*
  do not add a standard valence set of hydrogens to explicitly encoded atoms, even if the file format specification defines such a set. The most common application is for reading **SMILES** strings without the default hydrogen atoms. *nohadd* is a (slightly misleading) alias for this flag. This flag is independent of the generic hydrogen addition/removal processing option, which can be configured with the *hydrogens* attribute.

- *nometa*
  if this flag is set, it asserts that the file does not contain metal atoms. This is for example useful for reading **PDB** files which frequently possess ambiguous encodings such as *CA* for calcium or alpha carbon.

- *nometalh*
  suppress addition of hydrogens to metal atoms.

- *noradicals*
  assert that the file does not contain records with atoms that are radicals. This is a hint which is used for hydrogen addition, radical charging, and other operations.

- *pedantic*
  apply pedantic checking of file syntax rules. For some frequently abused file formats, such as **MDL** Molfiles or **PDB**, this may result in quite a percentage files being rejected for file format specification violations.

- *radicalcharger*
  Edit radicals which are typically formed by reading a file without formal atomic charge information by adding standard formal charges, for example replacing NR_4 with $N^{(+)}R_4$ and OR with $O^{(-)}R$. This only works reasonably well if the file contains a complete hydrogen set.

  - *readas2d*
    Force the interpretation of atomic coordinates as 2D, regardless of the file type encoding or presence of a third coordinate column, which may have been abused as an additional atom data store.
  - *readparity*
    If this flag is set, the parity fields in **MDL** Molfiles and derivatives are read and the data stored in property `A_LABEL_STEREO`. In accordance with **MDL** rules, this field is normally ignored, and stereochemistry decoded from wedge bonds and atom coordinates.
  - *sharedmap*
    If the file is memory-mapped, use a shared memory segment for the data. This can be useful if there are many processes accessing the same file for reading. This flag is only supported on Unix/Linux.
  - *simpleradicals*
    If this flag is set, the input file is assumed to contain only simple doublet radicals, if any. Any encoding of other, probably miscoded radical forms is changed to a doublet.
  - *tautoresolver*
    Perform a tautomer standardization on the read structure. This operation invalidates numerous atom and bond properties, such as coordinates, but in this special case all ensemble properties which were attached to the processed structure are retained, regardless of their sensitivity toward atom and bond changes. Tautomer resolution requires a complete hydrogen set, so either these must be present in the input file, or a suitable hydrogen addition mode must have been set on the file handle. The processing behind this input option is comparatively expensive. For normal input, when speedy input and maximum fidelity of the data to the original file is desired, this flag should not be set.
- *readkey*
  This attribute is only used in certain library configurations which have been configured to restrict read access to specific types of files. The key and data computed from the file name must together match the signature. Usually restricted applications have a compiled-in signature, and one or more read keys which enable read access to the same number of specific files.
- *readkeysignature*
  This attribute is used for certain library configurations which have been configured to restrict read access to specific files. This signature is required to verify the read access key.
- *readkeystatus*
  A read-only attribute which reports the access key status for a file for which a read key has been specified. It can be *unchecked*, *verified* or *error.*
- *readscope*
  This attribute controls which types of objects are read from a file, in case the file contains more than one object type. For example,. **MDL RXN** files can be read as en ensemble record stream, or as a reaction record stream. **CACTVS CBIN** files can be read as a multi-record stream of individual ensemble or reaction records, or as a single dataset with additional dataset properties. **CTX** files allow access to individual molecules or ensembles. The hierarchical **FDA SPL** format supports read modes for molecules, ensembles, and datasets. The default value for this attribute depends on the file format and is automatically updated whenever the

format is analyzed or changed. It is generally set to the most commonly used access variant for that format, for example reactions for **RXN** files and ensemble streams for **CBIN**, but it may also be set explicitly. Possible values are *none*, *mol*, *ens*, *reaction*, *dataset, biologics, hierarchy* and *auto*. In case a file format does not support a specific variant, the next supported type to the right in this sequence is automatically used. The *auto* mode performs a new content analysis for every record and use the most suitable scope. Examples where this is useful are **RDF** files with mixed structure and reaction records, or **RTF** documents which mix reaction and structure **OLE** objects. The *dataset* mode is potentially dangerous when reading large multi-record files which do not contain multiple smaller datasets. In that case, the whole file is interpreted as a single dataset, and that can lead to a large amount of memory being consumed.

- *record*
  The number of the next record to be read or written, starting with one. This value always refers to the current physical file, even if the file handle manages a virtual file. In case a virtual file is read, the *vrecord* attribute can be used to access the global record number. *rc* is an alias name for the attribute.

  It is possible to set this attribute in order to reposition the file pointer. In case the file is opened for output, and is not in *update* or *append* mode, this operation truncates the file. Repositioning while reading does not modify the file. It is not possible to position the file pointer any further to the rear of a file than immediately behind the end of the last existing record. In case of virtual files, a record setting implicitly changes the *vrecord* attribute, not the current physical file record.

  When setting the attribute, the special values *end* and *last* can be used to position the file pointer behind the last, or before the last record, and a negative value is interpreted as a backspace from the current position. The return value is the resolved record number.
- *recordtable*
  This is a read-only attribute. It returns a nested list of the attributes of the currently known record positions in the file. Every list element is itself a list which contains, in this order, the record number, the file offset, the line number (which is the same as the record number for binary formats), the *eor* type of that record, a boolean flag indicating whether the record is physically present in the disk file (0) or virtual (1)., and the original file name used to create the handle. In case of multi-file handles, this is not a constant over all records. In order to guarantee that all records of a file and their offsets are known, execute for example a **`molfile count`** command before querying the record table.
- *refcount*
  If the **Tcl** interpreter is using native **Cactvs** objects instead of string-based major object handles and integer-based minor object labels to identify toolkit objects, this returns the number of **Tcl** object references active for this object. The attribute is read-only.
- *references*
  Cross references of the file. This is a nested list of class **UUID**s and reference type tags.
- *replyto*
  The future recipient of a file. This is only set when the file has been extracted from a mail message or attachment. In order to send mail messages to specific destinations via the mail wrapper I/O module, this attribute may also be set.

- *resolution*
  A resolution value in DPI (dots per inch). The default value is 0, meaning that it is undefined. This information can be used by a couple of I/O modules, for example for reading structure data from image files by performing chemical **OCR** via the interface to the **OSRA** program.
- *returnformat*
  The name of the desired return format if the original file was received by mail. This is only set when the file has been extracted from a mail message or attachment.
- *scandata*
  This is a read-only attribute which reports statistics on the last `molfile scan` command. The returned data is a TCL dictionary with keys *start_time* (in seconds since 1970-1-1), *stop_time* (in seconds since 1970-1-1), *scan_time* (in seconds), *ens_read* (count of ensemble objects instantiated), *miniens_read* (count of Minimol objects decoded), *reactions_read* (count of reaction objects instantiated), *properties_read* (count of property records read), *ens_screened* (count of bit-screen filtering operations performed for substructure/superstructure searches), *reactions_screened* (count of bit-screen filtering operations performed for reaction matching), *records_examined* (count of records looked at), *records_matched* (number of matched records), *start_record* (record the scan started at), *end_record* (last visited record), *eof_reached* (boolean indicator whether the end of the file was reached), *max_mmap_used* (maximum used size of memory mapping arena), *max_mmap_requested* (maximum requested size of memory mapping arena), *records_skipped* (number of records which where skipped with need for re-synchronization), *records_repositioned* (number of records which were finished without the need for a re-synchronizing skip operation) *scores_computed* (the number of scoring function calls executed).
- *scoped*
  A boolean object visibility control flag. If set, and global control flag `::cactvs(object_scope)` is also set, the object is visible only in theTCL interpreter which set the scope flag and thus claimed it. Object list commands executed in other interpreters omit this object, and attempts to decode its handle in other interpreters will fail. The most common use of this feature is the hiding of persistent chemistry objects in scripted property computation functions.
- *selected*
  Flag indicating whether the object is selected. This attribute can be changed.
- *selection*
  This attribute is not a *molfile* handle attribute, but a flag attached to individual records. If queried, the return value is a list of all record numbers for which this flag is set. Using `molfile set` with a list of record numbers in any order to modify the attribute resets the current flags, and creates a new set. Modifying the attribute via `molfile append` adds selection flags without resetting the current selection. The selection flag can only be set for existing records. If an attempt is made to set the selection flag ahead of the currently known position set, the command scans the record structure (as in `molfile count`), which can be a problem in case of non-rewindable input. In order to facilitate resetting of selection flags, the virtual attribute *deselection* can be accessed as the inverse of the selection. Setting it to an empty list selects all records up to the end of the file (again this triggers automatic forward scanning, if necessary), and appending a list of records removes them from the selection. The default value of the selection flag for any record is *false*.

- *separator*
  A string containing one or more column separator characters. This is used for example by the structure and reaction table I/O module. The attribute is also set when a table with an auto-detected separator character was read via the file handle. The default separator is a single tab character.
- *sessionkey*
  A free-form string intended to be used to identify sessions.
- *shmid*
  In case the memory map arena of the file is in shared memory, this is the shared memory key as read-only value. If the file is not mapped into shared memory, or on platforms where memory mapping is not supported, the value is always minus one.
- *signature*
  The signature of a mail message. This is only set when the file has been extracted from a mail message or attachment.
- *similarityproperty*
  The name of the property which is used for bitvector similarity computation in file scans. Its default default value is controlled by the global variable **`::cactvs(default_similarity_property)`** and is usually either `E_SCREEN` or `E_QUERY_SCREEN`. If a file is opened that contains information about the similarity property set when the file was written (for example, **CBS** and **BDB** formats), this attribute is automatically set to the value stored in the file.
- *size*
  The file size in bytes as read-only data. In case it is not known, for example because the file is accessed via a special stream or a pipe, zero is reported.
- *sizehint*
  The expected maximum record count of the file. This attribute is used by some I/O modules to pre-allocate room in files with complex storage layout, in order to avoid the need for expensive re-organization during later record writes. The **CBS** format especially benefits from this information. File formats which are simple record sequences have no use for this information. A value of zero, which is the default, specifies an unknown future size. If the final size is not known exactly, it is generally preferable to overestimate it somewhat than to be slightly short.
- *statusflags*
  A list of boolean flags which describe the status of the machinery behind the I/O operations of this handle. All set flags are reported. When checking for the presence of a flag, make sure not to use simple string comparison, because other flags may also be set. While it is possible to change the flags, this is not a common operation, and if done carelessly can disrupt the I/O functionality of the handle. The older attribute name *flags* is still a valid alias. The following flags are commonly seen:

  - *append* - all file output is append to the end of the file, ignoring the current write pointer position.
  - *binary* - the file is binary, without a line structure.
  - *bzip2-compressed* - the file is accessed via a pipe to the *bzip2* program.

- *checkedbinary* - the file contents were checked to determine whether they contain non-ASCII characters.
- *edited* - the file contains virtual edited records, or virtual deletes.
- *fakeposition* - the file has no meaningful offset positions for the beginnings of records, the offset data structures contain other forms of access information
- *gzip-compressed* - the file is accessed via a pipe to the *gzip* program.
- *incomplete* - the last file record was not read completely. This can be intentional in file formats which support basic and extended data groups, or can be an indication of a non-critical decoder problem.
- *indexed* - the file is accessed via an index file with record positions, not directly.
- *nommap* - memory-mapping of the file contents is suppressed.
- *initialized* - an initialization function of in the associated I/O module has been called
- *locked* - there is currently a *flock()/lockf()* style file lock active on the file.
- *mapallocated* - the memory mapping arena for the file was allocated and filled via some read operation, not *mmap()ed*.
- *memlocked* - a mapping of the file are locked into memory and are not swapped out.
- *readable* - the file handle can be read from.
- *readonly* - the file has been opened for read-only access, without the possibility to switch the handle to a different mode.
- *remotefs* - the physical file resides on a non-local file system.
- *rewindable* - the file can be rewound if necessary
- *scratch* - the file is a scratch file and is automatically deleted when the file is closed.
- *shared* - the file contents reside in shared memory.
- *validcount* - the current number of known positions is known to correspond to the total of records in the file.
- *virtual* - the file is a virtual file build from multiple physical files.
- *ucs2-encoded* - the file is accessed via a pipe to the *iconv* program.
- *url* - the file is accessed via a URL, not a file system path.
- *updating* - the file is currently being updated
- *writeable* - the file handle can be written to.
- *xdr* - the file is associated with an `XDR` encoder or decoder structure.

- *style*
  A free-format string identifying a predefined attribute bundle for graphics-oriented file formats. This is currently supported for **CDX**, **CDXML**, **SKC** and **TGF**, where, for example, the *acs* value selects settings corresponding to the “ACS Journal” settings in *ChemDraw* or *ISISDraw.*
- *structurecolumn*
  This attribute is the numerical index of a column in table data files which are, for example, read by the structure table I/O module. The column is expected to contain a string notation for the basic structure object which is returned by a **`molfile read`** operation. This string is decoded and the content of the other columns is attached as property data to this object. Typically the content of the structure column is a SMILES, SLN or InChI string. A negative value of this attribute indicates that the presence of such structure data is not confirmed. In that case, an attempt is made to determine the structure column automatically, and the attribute is updated accordingly. However, setting it explicitly may still be required in case there are multiple column with structure data, or there are too many unreadable or `NULL` row entries to allow automatic determination.
- *subformat*
  A enumerated value which encodes the subtype of the main file format. The most common values are *mol2d*, *mol3d* and *mol0d*, to indicate structure records with 2D or 3D or no coordinates. The type *reaction* can be encountered for **RDF** and **CTX** files with reaction data, since these can also be structure files in other cases. The attribute is automatically set when a file is read,. For some formats a explicit specification of the attribute controls the output formatting, for example for all file formats which contain an **MDL** *ctab* block, which can store either 2D, 3D, or 0D information, but not simultaneously.
- *substructurescreen*
  The name of the property which is used for bitvector screening in filtering records for substructure matching. Its default default value is controlled by the global variable **`::cactvs(default_substructure_screen_property)`** and is usually either `E_SCREEN` or `E_QUERY_SCREEN`. If a file is opened that contains information about the screen property set when the file was written (for example, **CBS** and **BDB** formats), this attribute is automatically set to the value found in the file.
- *superstructurescreen*
  The name of the property which is used for bitvector screening in filtering records for superstructure matching. Its default default value is controlled by the global variable **`::cactvs(default_superstructure_screen_property)`** and is usually either `E_NO_HYDROGEN_SCREEN` or `E_NO_HYDROGEN_QUERY_SCREEN`. If a file is opened that contains information about the screen property set when the file was written (for example, **CBS** and **BDB** formats), this attribute is automatically set to the value found in the file.
- *template*
  The name of a template file to be used for output formatting. At this time, only the **RTF** I/O module uses this information. It switches between *de novo* **RTF** formatting and replacing chemistry tags in the template file. If this value is set to an empty string, no template is used.
- *timeout*
  The maximum number of seconds to spend in a **`molfile scan`** command. When the time is exhausted, the scan terminates after the respective current record has been cleanly processed by all query threads, even if the end of the file has not been reached. Setting the

attribute to zero, which is the default, allows an unlimited time to be spent on a query. Another function where the timeout value is used is in reading a record via an Internet connection, for example an *http* or *ftp* URL. If the timeout expires and the record has not been downloaded, an error results.

- *tooltip*
  If the toolkit was compiled with factory support, this is the tooltip of the object icon on a workbench. This attribute can be changed.
- *url*
  A read-only attribute with the URL in case the file is accessed via an Internet connection. If no such connection exists, the result is an empty string. If a URL has been set, this attribute may be indexed using the same fields as a URL property data item in order to retrieve URL components.

  The allowed field names are *hash*, *host*, *hostname*, *href*, *pathname*, *port*, *protocol*, *search*, *user*, *password*, *directory*, *file*, *ipaddr*, *lastmodified* and *mimetype*. Note that in this context the *port* field name is the port the file is transferred via the Internet connection, which generally is not the same as the listener port for remote requests (see **`molfile get`** attribute *port*). Likewise, the mimetype here is the MIME type as reported by the server, not the file MIME type defined by the file format handler module. Example:

  ```
  set ip [molfile get $fh url(ipaddr)]
  ```
- *user*
  This is a shortcut for the user name part of a file or virtual file addressed via an URL. For simple retrieval it is equivalent to the URL field attribute *url(user)*. For some I/O modules, for example the interface to access **MYSQL** tables as virtual structure files, a change of the user name has an effect and results in a re-authentication of the database and table access, which can result in different access permissions. For normal files accessed via a URL a change of the attribute is ignored after the file has been opened. Files that are not associated with an URL have an empty user name value.
- *uuid*
  An automatically generated UUID globally identifying the object. This attribute is read-only, different for every object, and not dependent on the contents or format of the disk file this object is associated with.
- *valencelevel*
  For files which support this concept, an indicator what kind of structure (stable, intermediate, MS ion, etc.) is stored in the file.
- *version*
  The file format version as a string. This attribute is set automatically when a file is opened for reading. If it is not set, files are generally read or written in the latest supported version. If a data file contains a known version indicator, input routines in some cases adjust to older encoding standards. The I/O modules of some file formats support the writing of old versions. An example are the **CDX** and **CDXML** modules, which in the context of file versions explicitly set to less than 8.0 do not write the *InterpretChemically* tag which is not understood by older ChemDraw releases.
- *versionuuid*
  The version **UUID** associated with this file as per its authorship attributes.

- *vline*
  The current virtual line count as a read-only attribute. For simple files, this is identical to the standard line count (attribute *line* or *lc*). However, for virtual files opened by means of the **`molfile lopen`** command, this attribute is the global line number in the virtual file, while *line/lc* refers to the line count within the current physical file. The attribute name *vlc* is an alias.
- *vrecord*
  The virtual record number of the next record to be read, starting with one. For simple files, this is identical to the standard record count (attribute *record* or *rc*). However, for virtual files opened by means of the **`molfile lopen`** command, this attribute is the global record number in the virtual file, while *record/rc* refers to the record count within the current physical file. The attribute name *vrc* is an alias.

  This attribute can be set and changing it results in repositioning of the file pointer, and potentially even a change in the active physical file.

  When setting the attribute, the special values *end* and *last* can be used to position the file pointer behind the last, or before the last record, and a negative value is interpreted as a backspace from the current position. The return value is the resolved record number.
- *width*
  The maximum width of a structure or reaction depiction in points (1/72 of an inch). This is only used for graphics-oriented formats, such as **CDX**, **SKC** or **EMF**. If the attribute is set to a negative value, which is the default, the size is indirectly controlled by the bond length and atom coordinates. In case this attribute is set to a positive value, and the depiction would exceed the maximum width or height, it is automatically scaled down proportionately.
- *writeend*
  An enumerated value indicating what kind of record end marker should be written on output, if the file format has such a concept. Possible values are *none*, *mol* and *block*. The default value is *block*, which translates into the standard record terminator for almost all file formats. The *mol* type is only significant for **CTX** format output. The *none* value can be useful if a programmer wants to add custom data to the end of a record and then writes an end marker himself, as it could be done without too much effort for example for an SD file.
- *writeflags*
  A collection of boolean flags controlling output details. When queried, this attribute returns a list of the names of all set flags. Modification of this flag supports the standard bit manipulation prefixes. The following flag names are currently recognized:

  - *none* - no flags
  - *computeprops* - attempt to compute properties in the write list if they are not yet present in the output objects.
  - *contracthydrogens* - whether to store displayed hydrogens as explicit atoms, or contracted and linked to their bond partner atom as a hybrid object. This option only applies to the **CDX** and **CDXML** formats. By default, the option is not set.
  - *miniheader* - keep the file header as concise as possible.

- *nohydrogenisotopesymbol* - if set, the default isotope symbols for heavy hydrogen (D and T) are not used in output. Instead, a simple H symbol is always written. Setting this flag does not suppress isotope output per se - if the file supports other means to define isotopes (e.g. **M ISO** lines in **MDL** files) these are still used.
- *multiwriter* - prepare the file to handle multiple simultaneous writers. The only file format I/O module which currently supports this is **BDB**.
- *noimplicith* - do not output hydrogen atoms which were added as implicit atoms.
- *nomoleculesplit* - do not split the ensembles into individual sub-records when this is normally required by the output format. This currently is only used by the **MDL RXN** and **RD** I/O modules. **RXN** files usually store every reaction component as a sub-record. This flag can be used to force writing of a single reagent and product record each, where each part may contain multiple un-separated components.
- *nopropertymapping* - always synthesize property descriptions, do not attempt to map them onto existing standard system definitions. The only module currently supporting this feature is the **PubChem** ASN.1 module.
- *nostereo* - do not write stereo information into the file, even if present in the output structures.
- *nostereoperception* - do not attempt to perceive stereochemistry from the available object data such as 2D coordinates and wedges, or 3D atomic coordinates, even if the file format normally requires this information.
- *omitct* - if the inclusion of a structure connectivity table is optional, this flag can be used to suppress the output this block.
- *pedantic* - perform pedantic output format checking, for example by refusing to write long lines in text formats which exceed the exact format specification, or refusing to write structures with more atoms than officially supported.
- *rawcoordinates* - do not perform any coordinate checking, scaling, and centring but write the coordinates exactly as they are currently stored.
- *recalcbaseprops* - if the output file content is a single property (for example `E_GIF` for **GIF** or **PNG** files, `E_EMF_IMAGE` for **EMF** and **WMF** files), force recalculation of this property before output.

- *supergroupexpansion* - If a file format can either be written with expanded or contracted superatom groups (specified as type *SUP* in property `G_TYPE` and group label in `G_NAME`), the default is to write them contracted. If this flag is set, the expanded form is used instead. This option affects few file formats (currently *cdx* and *cdxml*). It does not perform expansion of superatoms which are only present as a single pseudo atom in the ensemble by decoding their tag (see **`ens expand`** command to achieve this). Rather, it expects the full set of atoms of the expanded form in the ensemble, plus one or more properly set up group objects indicating the atoms of the expanded form of a functional group or fragment which are not shown in the contracted style. If these groups are present, only the first atom in any group is shown, with the `G_NAME` data as atom tag, which overrides all other label information. However, the output file still contains the hidden atoms and their data. Tools like *ChemDraw* use this data to support interactive group expansion utilizing the original layout coordinates of the previously hidden atoms and other information.

- *synchronous* - use synchronous writes for files which normally use buffering to increase performance, for example in the *bdb* format.

- *splitmol* - Split output into individual ensembles and write each molecular fragment as a separate record.

- *upgrade* - if this flag is set, and the format of a file is not of the most current version, but there is an upgrade function available in the support library, invoke the upgrade function to change the file layout to the most current version. The *bdb* module is the only one which currently supports this feature.

- *write0d* - write records without coordinates if possible

- *write2d* - write 2D records if possible

- *write3d* - write 3D records if possible

- *writearo* - write aromatic bonds instead of a Kekulé form if the file format supports this. An example where this makes sense are **SMILES** files. A counterexample are **MDL** *Molfiles* - you can enforce the encoding of aromatic bonds of non-query structures as the *aromatic* query bond type with this option, but that is technically incorrect and violating the format specification. Nevertheless, there are third party programs which require data in that format aberration for further processing.

- *writecolor* - write atom and bond colouring information if this is an optional part of the file format specification.

- *writeenzymes* - if the output data contains enzyme superatoms, include them in the output if that is an option. The **SDF3000** I/O module is an example for a module recognizing this flag.

- *writelabels* - write explicit atom labels, as defined in the attribute *atomlabelproperty*, if the file format supports it. This does not override the natural numbering of the written atom objects. It only applies to formats which support a parallel user-defined labelling scheme, such as **CDX/CDXML**.

- *writename* - write a structure name section if this is optional information in the output. An example are SMILES files.

- *writekey*
  This attribute is only used in certain library configurations which have been configured to restrict write access to specific types of files. The key and data computed from the file name must together match the signature. Usually restricted applications have a compiled-in signature, and one or more write keys which enable write access to the same number of specific files.

- *writekeysignature*
  This attribute is used for certain library configurations which have been configured to restrict write access to specific files. This signature is required to verify the write access key.

- *writekeystatus*
  A read-only attribute which reports the access key status for a file for which a write key has been specified. It can be *unchecked*, *verified* or *error*.

- *writelist*
  A list of properties that should be included in the output if the file format supports this. Standard properties defining basic connectivity etc. usually do not need to be listed because they are written out by default where needed. Normally, this list contains only ensemble- or reaction-level properties, like SD data fields. Properties listed both in the write list and the drop list are not written. By default properties listed here are not computed. If they are not already present in the output objects, they are omitted. The *computeprops* bit in the *writeflags* attribute can be used to automatically initiate a computation attempt. Still, if a computation attempt fails, the output of that property data is silently omitted.

- *x*
  If the toolkit was compiled with factory support, this is the *x* coordinate of the object icon on its workbench. This attribute can be changed.

- *y*
  If the toolkit was compiled with factory support, this is the *y* coordinate of the object icon on its workbench.This attribute can be changed.

The attribute list above is also referenced by the **`molfile set`** command. This is the reason why it contains information about the read-only status of the individual attributes. Only attributes that can be set can be addressed by the **`molfile set`** command.

For the use of the optional property parameter list argument, refer to the documentation of the **`ens get`** command.

Filters in the optional filter set must apply directly to the file object. Filters which operate on other object types are ignored.

Variants of the **`molfile get`** command are **`molfile new`**, **`molfile dget`**, **`molfile jget`**, **`molfile jnew`**, **`molfile jshow`**, **`molfile nget`**, **`molfile show`**, **`molfile sqldget`**, **`molfile sqlget`**, **`molfile sqlnew`**, and **`molfile sqlshow`**. These commands only work on property data and cannot be used to access attributes..

molfile getline

```
molfile getline filehandle ?skiprecord?
```

```
f.getline(?skiprecord=?)
```

Read a text line from the file, with repositioning of the file pointer. This operation is only possible on text files which have been opened for reading. The command is not frequently used, because it tends to disrupt the normal file record parsing.

If the *skiprecord* boolean argument is set, the file is positioned to the beginning of the next record after the line has been retrieved.

The command returns the line read. Line termination characters are removed.

molfile getparam

```
molfile getparam filehandle property ?key? ?default?
f.getparam(property=,?key=?,?default=?)
```

Retrieve a named computation parameter from valid property data. If the key is not present in the parameter list, an empty string is returned (**None** for **Python**). If the default argument is supplied, that value is returned in case the key is not found.

If the key parameter is omitted, a complete set of the parameters used for computation of the property value is returned in dictionary format.

This command does not attempt to compute property data. If the specified property is not present, an error results.

Example:

```
molfile getparam $fhandle F_QUERY_GIF format
```

returns the actual format of the data in that property, which could be a **GIF**, **PNG** or a bitmap format.

molfile hloop

```
molfile hloop filehandle objvar ?maxrec? body
f.hloop(function=,?maxloop=?,?variable=?)
Molfile.Hloop(filename,function=,?maxloop=?,?variable=?)
```

This command is functionally equivalent to the **molfile loop** command. The difference is that for the duration of the loop command hydrogen addition is enabled for the file handle. The original hydrogen addition mode of the file object is restored when the loop finishes.

molfile hread

```
molfile hread fhandle ?datasethandle/enshandle/#auto/new? ?recordcount?
molfile hread fhandle ?datasethandle/enshandle/#auto/new? ?parameterdict?
f.hread(?target=?,?parameters=?)
Molfile.Hread(filename,?target=?,?parameters=?)
```

This command is identical to the **molfile read** command, except that standard hydrogen addition is enabled for the duration of the command. The original hydrogen mode is reset when the command completes.

The **Python** class method is a one-shot command. The transient *molfile* created from the initialization items is automatically closed when the command finishes.

Example:

```
set eh [molfile hread "myfile.mol"]
```

This is a simple single-record structure input with hydrogen addition, using a file name instead of a file handle. The file is automatically opened and then close for the duration of the command.

molfile jget

```
molfile jget filehandle propertylist ?filterset? ?parameterdict?
f.jget(property=,?filters=?,?parameters=?)
Molfile.Jget(filename,property=,?filters=?,?parameters=?)
```

This is a variant of **`molfile get`** which returns the result data as a **JSON** formatted string instead of **TCL** or **PYTHON** interpreter objects.

The Python class method is a one-shot command. The transient dataset created from the initialization items is automatically deleted when the command finishes.

molfile jnew

```
molfile jnew filehandle propertylist ?filterset? ?parameterdict?
f.jnew(property=,?filters=?,?parameters=?)
Molfile.Jnew(filename,property=,?filters=?,?parameters=?)
```

This is a variant of **`molfile new`** which returns the result data as a **JSON** formatted string instead of **TCL** or **PYTHON** interpreter objects.

The Python class method is a one-shot command. The transient dataset created from the initialization items is automatically deleted when the command finishes.

molfile jshow

```
molfile jshow filehandle propertylist ?filterset? ?parameterdict?
f.jshow(property=,?filters=?,?parameters=?)
Molfile.Jshow(filename,property=,?filters=?,?parameters=?)
```

This is a variant of **`molfile show`** which returns the result data as a **JSON** formatted string instead of **TCL** or **PYTHON** interpreter objects.

The Python class method is a one-shot command. The transient dataset created from the initialization items is automatically deleted when the command finishes.

molfile list

```
molfile list ?filterlist=?
Molfile.List(?filters=?)
```

This command returns a list of the molfile handles currently registered in the application. This list may optionally be filtered by a standard filter list.

Example:

```
molfile list
```

lists the handles of all open *molfiles* in the application.

molfile lock

```
molfile lock filehandle propertylist/objclass/all ?compute?
f.lock(property=,?compute=?)
```

Lock property data of the file handle, meaning that it is no longer subject to the standard data consistency manager control. The data consistency manager deletes specific property data if anything is done to the file handle which would invalidate the information. Property data remains locked until is it explicitly unlocked.

The property data to lock can be selected by providing a list of the following identifiers:

- Property names
  Valid property instances on the file object are locked. If the boolean *compute* flag is set, an attempt is made to compute the property if it is not yet present. Otherwise, a request to lock non-existent data is silently ignored. It is not possible to lock individual property fields.
- *all*
  All valid file properties are locked. The compute flag is ignored.
- *molfile*
  This is an object class identifier. All property data which is controlled by the file major object and attached to the specified object class is locked. Since files do not incorporate minor objects, this identifier is equivalent to *all*.

The lock can be released by a **`molfile unlock`** command.

This command is a generic property data manipulation command which is implemented for all major objects in the same fashion and is not related to disk file locking. Disk file locks can be set or reset by modifying the *molfile* object attribute *lock*. This is explained in more detail in the paragraph on the **`molfile get`** command.

The return value is the original *molfile* handle or reference.

molfile loop

```
molfile loop filehandle objvar ?maxrec? body
f.loop(function=,?maxloop=?,?variable=?)
Molfile.Loop(filename,function=,?maxrecords=?,?variable=?)
for obj in f:
```

Execute a loop over the file. Objects are read from the file from the current file position onwards. The type of object read (usually ensemble or reaction, but in principle it could also be a table or dataset object) depends on the read scope of the file. In the **Tcl** variant, the handle of every object input from a file record is assigned to the specified **Tcl** object variable. Next, the **Tcl** script code in the body argument is executed. The body code typically uses the value of the variable to perform some operations with the currently read object. After the body code has been executed, the object which was just read is deleted, and the cycle is repeated, either until **EOF** has been reached on the file (the default), or the maximum number of records specified by the optional parameter has been reached, whichever comes first. In either case, no error is generated when the end of file has been reached. Setting the maximum record count parameter to an empty string, or to a negative value, results in the default processing style running until the end of the file.

For **Tcl** scripts, within the loop, the standard **Tcl** **`break`** and **`continue`** commands work as expected. If the body script generates an error, the loop is exited. If the loop code generates an error, the loop is terminated and the error reported. Programs should not expect that the same object handle value stored in the variable is reused in each iteration.

Since the input objects are automatically deleted after they have been processed, it is not required to delete them in the loop code. Deletion requests on the loop object executed within the loop are ignored. Any other operation on the structure object is allowed. The loop code may perform repositioning operations on the input file, but not close it.

The **Python** version of the loop method does intentionally have a different argument sequence for convenience. The function argument may either be a multi-line string (similar to the **Tcl** construct), or a function reference. Functions are called with the reference of the current loop object as single argument, and have their own context frame, so that the specification of a reference variable is not generally useful in that call style, though is is allowed. For string function blocks the code is executed in the local call frame, and the variable with the current object reference is visible locally. Script code blocks must be written with an initial indentation level of zero. Within the **Python** functions, the normal *break* and *continue* loop control commands cannot be used to to scope limitations. Instead, the custom exceptions *BreakLoop* and *ContinueLoop* can be raised. These are automatically caught and processed in the loop body handler code.

In **Python**, there is also an object iterator so that simple loops over structure file contents can be written with a **`for`** statement. The *molfile* object iterator is of the *self* style (i.e. there is one per *molfile*, these are not independent objects), so nesting them is not possible on the same dataset. There is no distinct *hloop* iterator, but that can be emulated by setting the *hydrogens* attribute on the *molfile* object.

Python object loop constructs and their peculiarities are discussed in more detail in the general chapter on **Python** scripting.

The return value is the number of processed records.

Example:

```
set th [table create]
table addcol $th E_NAME
table addcol $th E_WEIGHT
molfile loop $myfile eh {
   table addrow $th #auto end [list [ens get $eh E_NAME] [ens get $eh E_WEIGHT]]
}
```

This sample loop successively reads all records from the file and stores the ensemble handles in variable *eh*. In the loop body, the handle is used to extract name and molecular weight information from the structure and store it in a table object.

molfile lopen

```
molfile lopen filelist ?mode? ?attribute value?...
molfile lopen filelist ?mode? ?attributedict?
Molfile(filenamesequence,?mode=?,?attribute=?)
Molfile.Open(filenamesequence,?mode=?,?attributes=?)
Molfile.Lopen(filenamesequence,?mode=?,?attributes=?)
```

Open a list of files as a virtual file. The files identified by the file list items are implicitly concatenated in the list order. In addition to normal files, the standard set of special input types such as **URL**s, pipes, **Tcl** file handles or standard channels may be used. This command returns a single file handle, regardless of the number of input files passed as parameter.

A file list can only be opened for read operations on input objects. Writing, appending, updating or string input are not supported.

Most input file operations can be performed on virtual files. One important exception is currently file scanning with query expressions. This only works for lists of standard sequential files, not files which contain optimized query layouts, such as the native **CACTVS CBS** and **BDB** file formats. These can only be used as a single file for `molfile scan` commands. However, simple structure input is possible across file boundaries even with these formats.

The rest of the options are processed in the same way as the standard `molfile open` command.

In the **PYTHON** interface, there is no distinction between the `lopen` and `open` commands, because it can be unequivocally established whether the filename argument is a sequence (tuple or list of filenames), or a single file name. The interpretation is performed according to the argument type. The **PYTHON** command always uses a file attribute dictionary, not a keyword/value argument set.

Example:

```
set fhandle [molfile lopen [lsort [glob *.mol]]]
```

molfile max

```
molfile max filehandle property ?filterset?
f.max(property=,?filters=?)
Molfile.Max(filename,property=,?filters=?)
```

Scan the file for the maximum value of the specified property from the current read position to the end of the file. If no error occurs, the file is at end-of-file after the end of the command.

If a filter set is provided, it is applied to the objects read from the file during the scan, not the molfile object proper. Objects which do not pass the filter are ignored.

The property may correspond either to a data column in the file, or to a computable property on the structure or reaction objects read during the scan. Read objects are transient and automatically discarded. The property argument may contain a field specification, and in that case, only the field value is compared.

The maximum value determination uses the standard property comparison function associated with its data type. For properties which are implicitly defined during file I/O, an explicit property definition with a correct data type may be beneficial. For example, when testing the values of an SD data field, by default the data is read as an implicitly created string property. If the field content is actually an integer, the comparison as a string value does not yield the same results as when the data is compared as an integer. For file formats which encode a proper data type of its contents this is not necessary.

The return value is the maximum property or property field value found, or an empty string if no input was processed.

The **PYTHON** class method is a one-shot command. The transient *molfile* created from the initialization items is automatically closed when the command finishes.

molfile metadata

```
molfile metadata filehandle property ?field ?value??
f.metadata(property=,?field=?,?value=?)
```

Obtain property metadata information, or set it. The handling of property metadata is explained in more detail in its own introductory section. The related commands **`molfile setparam`** and **`molfile getparam`** can be used for convenient manipulation of specific keys in the computation parameter field. Metadata can only be read from or set on valid property data.

Valid field names are *bounds*, *comment*, *info*, *flags*, *parameters* and *unit*.

molfile min

```
molfile min filehandle property ?filterset?
f.min(property=,?filters=?)
Molfile.Min(filename,property=,?filters=?)
```

Scan the file for the minimum value of the specified property from the current read position to the end of the file. If no error occurs, the file is at end-of-file after the end of the command.

If a filter set is provided, it is applied to the objects read from the file during the scan, not the molfile object proper. Objects which do not pass the filter are ignored.

The property may correspond either to a data column in the file, or to a computable property on the structure or reaction objects read during the scan. Read objects are transient and automatically discarded. The property argument may contain a field specification, and in that case, only the field value is compared.

The minimum value determination uses the standard property comparison function associated with its data type. For properties which are implicitly defined during file I/O, an explicit property definition with a correct data type may be beneficial. For example, when testing the values of an SD data field, by default the data is read as an implicitly created string property. If the field content is actually an integer, the comparison as a string value does not yield the same results as when the data is compared as an integer. For file formats which encode a proper data type of its contents this is not necessary.

The return value is the maximum property or property field value found, or an empty string if no input was processed.

The **PYTHON** class method is a one-shot command. The transient *molfile* created from the initialization items is automatically closed when the command finishes.

molfile mutex

```
molfile mutex filehandle mode
f.mutex(mode)
```

Manipulate the object mutex.

During the execution of a script command, the mutex of the major object(s) associated with the command are automatically locked and unlocked, so that the operation of the command is thread-safe. This applies to toolkit builds that support multi-threading, either by allowing multiple parallel script interpreters in separate threads or by supporting helper threads for the acceleration of command execution or background information processing.

Going beyond this automatic per-statement protection, this command locks major objects for a period of time that exceeds a single command. A lock on the object can only be released from the same interpreter thread that set the lock. Any other threaded interpreters, or auxiliary threads, block until a mutex release command has been executed when accessing a locked command object. This command supports the following modes:

- *lock*
  Increase the recursive mutex lock count on the object. The command returns the current lock count after the command, excluding the transient single-command lock.
- *reset*
  Release all persistent locks on the object, if any exist.
- *test*
  Return the current persistent lock count on the object. This excludes the transient per-command lock.
- *unlock*
  Decrease the recursive lock count on the object. The command returns the current lock count after the command, excluding the transient single-command lock. Unlocking an object which has not been persistently locked results in an error.

There is no *trylock* command variant because the command already needs to be able to acquire a transient object mutex lock for its execution.

The command returns the current lock count.

molfile need

```
molfile need filehandle propertylist ?mode? ?parameterdict?
f.need(property=,?mode=?,?parameters=?)
```

Standard command for the computation of property data, without immediate retrieval of results. This command is explained in more detail in the section about retrieving property data.

The return value is the original file handle or reference.

Example:

```
molfile need $fhandle F_AVERAGE_ATOM_COUNT
```

molfile new

```
molfile new filehandle propertylist ?filterset? ?parameterdict?
f.new(property=,?filters=?,?parameters=?)
Molfile.New(filename,property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **`molfile get`** command. The difference between **`molfile get`** and **`molfile new`** is that the latter forces the re-computation of the property data, regardless whether it is present and valid, or not.

The **Python** class method is a one-shot command. The transient *molfile* created from the initialization items is automatically closed when the command finishes.

molfile nget

```
molfile nget filehandle propertylist ?filterset? ?parameterdict?
```

```
f.nget(property=,?filters=?,?parameters=?)
Molfile.Nget(filename,property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **`molfile get`** command. The difference between **`molfile get`** and **`molfile nget`** is that the latter always returns numeric data, even if symbolic names for the values are available.

The **Python** class method is a one-shot command. The transient *molfile* created from the initialization items is automatically closed when the command finishes.

molfile nnew

```
molfile nnew filehandle propertylist ?filterset? ?parameterdict?
f.nnew(property=,?filters=?,?parameters=?)
Molfile.Nnew(filename,property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading object data and attributes. It is explained in more detail in the section about retrieving property data.

For examples, see the **`molfile get`** command. The difference between **`molfile get`** and **`molfile nnew`** is that the latter always returns numeric data, even if symbolic names for the values are available, and that property data re-computation is enforced.

The **Python** class method is a one-shot command. The transient *molfile* created from the initialization items is automatically closed when the command finishes.

molfile open

```
molfile open filename ?mode? ?attribute value?...
molfile open filename ?mode? ?attributedict?
Molfile(filenamesequence,?mode=?,?attribute=?)
Molfile.Open(filenamesequence,?mode=?,?attributes=?)
```

This command opens a structure file or other input source for input or output. The *filename* argument may be any of:

- A disk file

  This is the most common case. File names may be absolute or relative. On the Windows platform, the path naming follows the **Tcl** convention, with backslashes replaced by forward slashes, and optional drive letters, in the same way as the standard **Tcl `open`** command. Tilde substitution is also supported and built into the command. In case a file name could possibly collide with a reserved name, the file name can be prefixed with ./ in order to force interpretation as a file name. File name expansion can be conveniently performed by means of the standard **Tcl `glob`** command. File names must currently be spelled in the 8-bit ISO8859-1 character set. Unicode file names are not yet supported. On Unix platforms, named pipes and sockets may also be opened with this command.

  Examples:

  ```
  molfile open ./stdout r
  molfile open ~theuser/data/newleads.sdf
  molfile open C:/temp/calicheaamycin.pdb w
  ```

- A standard channel

  The file names *stdout*, *stderr* and *stdin* are reserved and connect the file handle to a standard I/O channel. *stdout* and *stderr* can only be opened for output, and *stdin* can only be read from. The character '-' (minus) is an alternative name for standard input.

  Example:

  ```
  molfile open stdout w format mdl
  molfile open ./stdout
  ```

  The first line opens an **MDL** file for output on standard output. The second sample line opens the file in the current directory which is named "stdout" for input. By prefixing file names with directory information any file with a reserved name can be opened as standard file.

- A scratch file

  The name *scratch* is reserved as the name of a generic scratch file. The file is initially opened for writing, but may be switched to input later by a **`molfile toggle`** command. The magic filename is translated into the name of a platform-specific temporary file. Every invocation of this command variant generates a new scratch file, with a different name. The true file name can be obtained with an attribute query:

  ```
  set fh [molfile open scratch]
  set name [molfile get $fh name]
  ```

  Scratch files are automatically deleted when they are closed, or when the program exits.

- A pipe

  If a file name starts with a vertical bar character "|", a pipe is opened from (in read mode) or to (write mode) the commands listed after the bar.

  Example:

  ```
  molfile open "|gzip >thefile.sdf.gz" w format mdl
  ```

  When the file is closed, the pipe and all programs connected to it are automatically shut down. Pipes cannot be rewound, or switched from input to output and vice versa.

- An **URL**

  The **CACTVS** toolkit supports reading from various types of URLs. Currently, the schemes *ftp*, *http*, *file* and *gopher* are supported. *file* URLs are just another notation for normal disk files, as described above. From among the other URL schemes, only *ftp* and *http* connections may be opened for writing. The support for *ftp* URLs includes username and password components. If the server side supports it, passive *ftp* is the preferred mode. *Http* connections opened for writing use the **PUT** *http* command, which often is not activated in standard Web server set-ups and may therefore be of limited practical usefulness. URL connections can be rewound and backspaced, but this is costly because the existing connection has to be disconnected and the initial data from the beginning of the file to the desired position needs to be re-transferred and discarded.

  Examples:

  ```
  set fh [molfile open http://www.yourcompany.com/repository/jcamp/ir1.jcp]
  molfile open ftp://yourid:yourpasswd@ftp.yourcompany.com/upload/ideas.sdf
  ```

- A directory

  If the target is a directory, all files in the directory are scanned. Those files which were identified as structure data files by any of the built-in or currently loaded I/O module extensions are concatenated to a virtual file which comprises all individual files. The order in which the files are concatenated is largely unpredictable, because it is defined by the order of the file name entries in the directory, and not any alphabetic sort criterion. The files may be of different formats, and may be any mixture of single-record and multi-record files. Subdirectories of the opened directory are not entered by default, but this may be activated by appending a ‚d‘ character to the open mode. Directories may only be opened for reading.

  Example:

  ```
  set fh [molfile open .]
  set fh [molfile open $mydir rd]
  ```

  The second example opens not only perceived structure files in the source directory, but also in all subdirectories thereof.

- A string

  The **CACTVS** toolkit can read most file formats directly from a string. There is no need to write structure data which was obtained as a string image to a temporary file to decode it. Data strings are opened as structure file with mode 's'. Only input is possible, but navigation within the string with **`molfile rewind`** etc. works as expected. The complementary **`molfile string`** command can be used to generate a string image of a file record.

  Example:

  ```
  set fh [molfile open $thedatablob s]
  set eh1 [molfile read $fh]
  set eh2 [molfile read $fh]
  molfile close $fh
  ```

- A **TCL** file or socket handle, or a **PYTHON** file reference

  Any file name beginning with *file* or *sock*, and where the rest of the file name is a sequence of digits, are interpreted as references to **TCL** file handles.

  Example:

  ```
  set tcl_fh [open thefile.txt w]
  set cactvs_fh [molfile open $tcl_fh w]
  ```

  A **TCL** handle can only be accessed by this command in a mode which is compatible to the mode it was opened with, i.e. it is not possible to write to a file via a **`Tcl`** handle if it was opened for reading. If a structure file coupled to a **TCL** handle is closed with a **`molfile close`** command, the **TCL** handle remains valid, and my be used freely once the association to the structure file I/O object is broken. Closing the **TCL** file handle while the piggybacked structure file handle is being used is illegal. No input, output or positioning should be performed on the **TCL** handle with standard **TCL** commands while it is being referred to by a *molfile* object.

  In the **PYTHON** interface, the same mechanisms apply, except that the argument is a **PYTHON** file handle object.

The **Tcl** handle functionality is not available on Windows, because on this platform **Tcl** internally uses Windows handles for I/O, while the **Cactvs** toolkit builds on standard Posix C library **`FILE*`** pointers.

- A virtual file

  Some I/O modules implement access to a variety of information sources as a virtual file, which has neither a presence on the local disk, nor is one of the standard magic file names or access methods. Such virtual file names are by convention written with pointed brackets.

  Example:

  ```
  set fh [molfile open <pubchem>]
  ```

  This command loads the **PubChem** virtual file access module, and returns a handle which may be used in a similar fashion as, for example, a handle to a huge local SD file. Depending on the I/O module, various operations on the handle may be optimized to be performed remotely. For example, the **PubChem** module offloads as many query operations of **`molfile scan`** commands as possible to the NCBI computers and downloads result structures only if they are needed as results, or query sub-expressions were specified which cannot be processed by the NCBI system.

The first optional parameter is the file access mode. It may be one of:

- r
  Open for reading, but with the option of later changing the mode to writing or appending. This is the default.

- rt
  As above, but automatically start a thread which immediately starts gathering file status information, such as the record count and record positions. This mode can be useful when operations, such as reading data for display, are to be commenced immediately, but ultimately overall record count information needs to be displayed, which can take a while to collect for larger files. The status thread is only started for rewindable files, and has no effects on files which directly provide record index and total record count information. Operations which would duplicate the efforts of the statistics thread, such as **`molfile count`**, are automatically blocked until the thread has completed,, and then directly use its results. Operations which change the nature of the access to the file, or its record contents or positions, silently terminate the status thread.

- ro or rot
  Open for read-only. If a file is opened in this mode, it is not possible to switch to write access later via a **`molfile toggle`** command. If the file permissions do not allow write access, the standard ‘r’ mode automatically falls back to this variant. Mode ‘rot’’ is also possible and additionally starts a file status thread (see mode ‘rt’).

- w
  Open for writing. If the file exists, it is overwritten. If not overridden by an explicit format specification, the file format is inferred from the suffix of the file name, if possible.

- a
  Open for appending. If the file exists, new data written to it is appended. If not overridden by an explicit format specification, the file format is inferred from the suffix of the file name, if possible. Not all file formats support appending.
- u
  Open an existing file for updating, i.e. the replacement of specific records. Not all file formats support this mode. It is generally useful for database-style formats such as **CACTVS BDB** and, to a limited degree, **CBS**. It can also be used for simple record sequence files like SD, though in this case it can be inefficient because a lot of data copying may be required to adjust the file layout. For single-record file formats, this command is not useful, and multi-record files which are not simple record sequences and for which the I/O module does not provide a special function, this mode is not supported.
- s
  Open string image of a file. If the mode is used, the file name is interpreted as an in-memory image of a structure file in any of the formats the toolkit understands, and not as a file name, URL, or any of the other types of input objects. Binary file formats may be used with this command.
- p
  Open in pipe reader mode. The input is expected to be a pipe or socket, where sporadically new data is posted. If an attempt is made to read from the file, a check is made if any data is present. If no data is waiting, the input command immediately returns without blocking. At a later time, new data may be present and the input succeed. If just a single byte of data is present on a pipe input channel, the read routine hangs on until the record for which input has begun has been read completely.
- R, Ro, Rot
  Open the file for reading and infer the format of the file from the suffix alone, without actually attempting to read the initial section of the file contents, which is the default method to determine its format. This mode can be useful in case the data contains text with embedded structure data, where the plain text is read by scripted commands and the occasional embedded structure or reaction record is to be extracted by means of **`molfile read`** commands. For such files, an automatic format detection would fail. The ‘o’ and ‘t’ flags may also be appended, and have the same meaning as in the standard ‘r’ mode.

For some files and file formats, two more mode characters have meaning if appended to the primary mode: They are silently ignored if the file argument or file format do not support them.

- d
  Recursive opening. This is for example useful when opening a directory as a pseudo file for input. If this flag is set, the all files recursively found under the specified directory form the virtual file, not just the files directly located under the specified directory.
- f
  Fast mode. The file is opened for maximum performance, taking chances with respect to data integrity in case of program or computer crashes, etc. One file format where this flag is supported is the Cactvs Data Archive (**CDA**) format.

The remaining parameters of the **`molfile`** command are optional keyword/value pairs, or alternatively a single dictionary with the same function. The processing of these parameters is exactly the same as in the **`molfile set`** command.

In the **Python** interface, there is no distinction between the **`lopen`** and **`open`** commands, because it can be unequivocally established whether the filename argument is a sequence (tuple or list of filenames), or a single file name. The interpretation is performed according to the argument type. The **Python** command always uses a file attribute dictionary, not a keyword/value argument set.

Example:

```
set fhandle1 [molfile open thefile.pdb]
molfile set $fhandle1 hydrogens add nitrosyle ionic
set fhandle2 [molfile open thefile.pbp r hydrogens add nitrostyle ionic]
```

The first two lines and file final line perform exactly the same task: Open an input file, and set up input flags so that a complete set of hydrogens is added, and nitro groups and similar groups are converted to an ionic (as opposed to pentavalent) representation.

When a file is opened for reading, its format is automatically determined. Do not use the *format* attribute except under very special circumstances.

The command returns the file handle or reference of the opened input file. This is the handle which is required by most other **`molfile`** commands which refer to an opened file.

Depending on the encoding of the opened file, the actual access mode to the file may be different than expected. In case a disk file is compressed with *gzip* or *bzip2*, the file is opened via a pipe to the responsible decompressor program. Likewise, an **UCS-2** encoded file is opened via a pipe to the *iconv* program which converts the contents to the **UTF-8** encoding. Files which are opened indirectly via such helper pipes have different access characteristics than directly addressed files. For example, backspacing is expensive, because the pipe has to be closed, re-opened, and the data stream skipped to the desired position. This takes much longer than simply repositioning a file pointer.

molfile peek

```
molfile peek filehandle
f.peek()
Molfile.Peek(filename)
```

This is a convenience command which combines three operations: Read the next record (**`molfile read`**), discard whatever object is read by the command as configured by the file handle settings (**`ens/reaction/dataset delete`**), and backspace by one record (**`molfile backspace`**).

The purpose of this command is to learn more about the contents and characteristics of the file by performing a full parse of the next record. One of the most common applications of the command is to detect the field structure (such as SD data fields) of that record before the read. The detected field set is consequently the return value of the command, equivalent to a **`molfile get`** ***`filehandle`*** **`fields`** statement.

This command can only be used on files which can be backspaced, or at least rewound and skipped forward to the last position. It cannot be used on files not opened for input, on empty files, or files which are at **EOF**. In all these cases, an error results.

The **Python** class method is a one-shot command. The transient *molfile* created from the initialization items is automatically closed when the command finishes.

molfile properties

```
molfile properties filehandle ?pattern? ?noempty?
f.properties(?pattern=?,?noempty=?)
```

Generate a list of the names of all properties attached to the *molfile* object. Optionally, the list may be filtered by a string match pattern.

In most cases, this list is empty. Only structure file properties, such as **F_COMMENT**, etc., are listed, but no object attributes, such as *readflags*, *nitrostyle*, etc. Few file formats support the concept of storing file-level properties, and therefore an empty property set is usually reported. Since file objects do not contain minor objects, and currently cannot be a member of other major objects such as datasets or reactions, no properties belonging to other classes except file objects are ever listed.

If the *noempty* flag is set, only properties where at least one data element is not the property default value are output. By default, the filter pattern is an empty string, and the *noempty* flag is not set.

The property list may become modified by input operations. In some cases, the defined file-level properties may vary with the record position, or may become only available only after the first input operation, not immediately after opening the file.

The command may be abbreviated to **props** instead of the full name **properties**.

Example:

```
set plist [molfile properties $fhandle]
```

molfile purge

```
molfile purge filehandle propertylist/molfile/all ?emptyonly?
f.purge(?properties=?,?emptyonly=?)
```

Delete property data from the *molfile* object. Only *molfile* property data may be deleted with this command (these usually have a `F_` prefix). Molfile attributes are not deletable.

If the optional flag is set, only file property values which are identical to the default of the property are deleted. By default, or when this flag is 0, properties are deleted regardless of their values. In case a listed property is not present, or not a file property, the request is silently ignored, but using property names which cannot be resolved leads to an error. If the object class name *molfile* is used instead of a property name, all file-level property data is deleted from the molfile object.

The command returns the original *molfile* handle or reference.

Example:

```
molfile purge $fhandle F_COMMENT
molfile purge $fhandle all
```

The first command deletes a specific property, the second command deletes all file property data associated with the handle.

molfile putline

```
molfile putline filehandle ?lines?
f.putline(?line?,...)
```

Write user-specified string lines to a file, bypassing the normal record writing mechanism. This operation is only supported on files which are opened for output and contain text data. The lines should not contain end-of-line characters. These are automatically supplied depending on the file object configuration set in the *eolchars* attribute.

The command returns the original file handle or reference.

molfile read

```
molfile read fhandle ?datasethandle/enshandle/#auto/new? ?recordcount?
molfile read fhandle ?datasethandle/enshandle/#auto/new? ?parameterdict?
m.read(?target=?,?parameters=?)
Molfile.Read(filename,?target=?,?parameters=?)
```

This important command reads chemistry objects from a structure or reaction file. The type of objects returned depends on the read scope of the file. They can be ensembles, reactions, or datasets. Read scope *mol* returns single-molecule ensembles, but (with I/O modules supporting this feature) reads only individual molecules into the output ensemble, splitting a multi-molecule file data ensemble if necessary. The return value of the command is a list of the handles or references of all objects which were generated, except when the *#auto* dataset creation method was used, or an unlimited number of objects was read into a dataset. In that case, the recipient dataset handle or reference is returned.

By default, the returned objects are not a member of any dataset. If a dataset handle is passed as fourth parameter, the returned objects are appended to that dataset if possible. The special value *#auto* or *new* creates a new dataset as container. This is equivalent to using the nested statement **`[dataset create]`** as dataset handle argument. If the fourth parameter is an ensemble handle, and the object read from the file is also an ensemble, the read data is stored in the shell of the old ensemble, after all old ensemble data has been deleted. Its object handle remains unchanged, as is its dataset membership. The reuse of reaction handles is currently not supported. This parameter can be skipped by specifying an empty string.

In addition to passing an empty string, or a simple dataset or ensemble handle or reference, as the fourth command argument, a list/tuple consisting of a handle or reference and a modifier flag set can be specified. The only flag value which is currently recognized is *checkroom*. If that flag is set, and the input objects are to become members of a dataset with enabled maximum size or insertion mode control, a test is made whether the dataset has sufficient room to allow the insertion of the new object(s), or whether a suitable alternative action is configured to handle the read object in a different fashion, such as discarding it. If that is not the case, the command returns immediately, without performing any input, and returns an empty string (**`None`** for **PYTHON**). If the test succeeds, the input operation is atomic, since the dataset is locked for the full duration of the command, so that no other threads can manipulate its status between the initial check and the file input result object transfer.

The final optional parameter is either a single argument specifying the number of objects which should be read, or a dictionary with key/value attributes. The default is equivalent to passing a simple numerical value of one, in the first, simple format. In order to read until the end of the file, the special value *all* may be used instead of a numerical count. With an *all* parameter value, the input operation is finished when no more data is available on the file. Until this condition is met, an unlimited number of records is read. No error is generated when **`EOF`** is met. There are also no **`EOF`** errors reported if a numerical record count of more than one was specified, and at least one object could be successfully read. Another magical value of the simple argument form is *batch*, which is substituted by the batch record set size configured on the *molfile* handle (see **`molfile get/set`**).

In the second form of the final parameter, an attribute dictionary is persistently applied equivalent to a **molfile set** command before the input commences. Standard file handle attributes and an input limit may be both set in parallel by using the special attribute name *limit* as part of the dictionary. It is only recognized in this context, but not with **molfile set** or **molfile string**. The allowed values of the *limit* attribute are the same as in the simple command variant.

The command raises an error if input could not be completed, regardless whether the reason is a file syntax error, or simple **EOF** (but see above for exceptions). If an input error occurs, the **EOF** attribute of the file handle should therefore be checked in order to distinguish between these two conditions. In case the input file was opened for pipe reading (mode ’p’), or is connected to a **TCL** channel, an **EOF** report may only indicate that no current data is available on the pipe or **TCL** channel, but it could still arrive at a future point in time.

Examples:

```
if {[catch {molfile read $fhandle} ehandle]} {
   if {![molfile get $fhandle eof]} {
      puts “Error: $ehandle”
   }
} else {
   puts “Read [ens get $ehandle E_NAME]”
}
```

The prototypical snippet above shows the input of the next ensemble record from a previously opened file, with proper error checking.

```
molfile read “acd.sdf” [dataset create] all
```

This sample command reads a complete input file (we are using the single-operation feature of the **molfile** command to open and close the file *acd.sdf* automatically for the duration of this command) into a newly created dataset in memory. Reading huge datasets is of course not necessarily a good idea without large amounts of **RAM**. On typical current workstations, 10.000 or 20.000 compounds are no problem, but beyond that the risk of running out of memory is a real problem.

In default mode, hydrogens are *not* automatically added to the read items with the exception of file formats where a clearly defined hydrogen set is implied, like **SMILES**, but not **MDL** *molfiles*). This is probably the most common problem developers run into when using this command. Generally, Cactvs wants to operate on hydrogen-complete structures, and its internal file formats use explicit hydrogen encoding. Working with hydrogen-incomplete structures is possible, and sometimes useful, but can lead to unexpected artifacts like radical centers on atoms with missing hydrogens. In order to continue with a standard hydrogen set, the most common options are:

- Use **molfile set** to change the *hydrogens* attribute to a suitable automatic hydrogen addition mode. The **molfile open** command can also configure this attribute directly in a single statement, or you can use the attribute dictionary form of this command for the same purpose.
- Use **molfile hread** instead of **molfile read**
- Execute a **ens/reaction/dataset hadd** command after the input object is returned and before processing the read objects further.

molfile ref

```
Molfile.Ref(identifier)
```

Python only method to get a *molfile* reference from a handle or another identifier. For *molfiles*, other recognized identifiers are *molfile* references, integers encoding the numeric part of the handle string or the **UUID** of the molfile object.

molfile rename

```
molfile rename filehandle srcproperty dstproperty
f.rename(srcproperty=,dstproperty=)
```

This is a variant of the **molfile assign** command. Please refer the command description in that paragraph.

molfile reorganize

```
molfile reorganize filehandle
f.reorganize()
```

This command only has an effect for file formats for which the I/O module provides a reorganizer function. This function typically optimizes and compacts the file for input and queries, and should usually be called after all records have been written. Writing to a reorganized file is typically at least initially slower than writing to a file which has not been processed.

The function returns a boolean value indicating whether any reorganization has actually been performed. In case the command is applied to a file which is not writable, an error results.

molfile rewind

```
molfile rewind filehandle
f.rewind()
```

Reposition the file before first record, and clear all error status information. If the file is already at the first record, and no error condition is set, this command does nothing.

Not all file channels can be rewound, and for some which can, it can be an expensive operation. For example, standard input or pipe input channels are not rewindable, and an **FTP** URL channel has to be closed and re-opened.

Rewinding a virtual file set positions the file pointer before the first record of the first file in the set.

Standard text-stream style *output* files can be rewound, too. This effectively truncates them. Files which are opened for appending are truncated to their original length.

Rewinding is not necessary in all cases. The **molfile scan** command automatically rewinds the input file if it is at **EOF** at the begin of a scan.

The return value of the command is the original file handle or reference.

molfile rewrite

```
molfile rewrite filehandle recordlist propertylist ?values? ?query? ?callback?
m.rewrite(records=,properties=,?values=?,?query=?,?callback=?)
```

This command updates specific property fields in a file, without rewriting the complete record. This is only supported if the file was opened for writing or updating, and the I/O module for the format of the file supports this operation by a special function. This typically limits the applicability of this command to database-style file formats such as **Cactvs CBS** and **BDB**.

The record list parameter is either a simple sequence of numerical records, with one as the first file record, or one of the special values *all* (all file records are updated), *current*, *next*, *previous* (the indicated record is updated), or a table handle, optionally followed by a table column name. In the last case, the table is expected to contain the data for rewriting, and in case a column name is specified, that column should contain the applicable record numbers. If the table version is selected without a record column, the file records from one to the number of table rows are updated. None of the special values can be combined with the simple numerical record sequence style. If the parameter is a numerical record sequence, the order of the records is significant.

The values sequence can be empty, or it must match the length of the property list. In the latter case, every specified value must be a valid value for the property in the same list index position. Note that while it is possible to manipulate multiple records in one step with this command, it is not possible to assign a different set of values to the data fields for each processed record. For this operation, multiple rewrite statements must be issued. If the value list is absent, or empty, the values are recomputed from the structure or reaction object that is temporarily read from the file record for this purpose. This is a useful feature in case the computation function for a computable property has changed. In case the record list references a table instead of a numerical record list or a magic record name, the value list is ignored. Instead, the table is expected to contain table columns which match the properties in the list, but not necessarily in the same column order, or containing exclusively the properties in the list.

The optional query argument is a query expression in the same style as used in the **`molfile scan`** command. If a filter expression is supplied, only records which match the expression are changed. Non-matching records are skipped. In case no filter is used, all records selected by the record list are processed

After processing, the file pointer is on the last processed record.

If the name of a callback procedure is specified in the **Tcl** interface, it is called after each processed record. The **Tcl** procedure arguments depend on the processing mode. In case of table-based processing, the arguments are the table handle, the current table row, the file handle and the current file record. In the **Python** interface, the callback is either a function name given as string, or a function reference.

This command is not fully implemented yet. **CBS** files currently only support re-computation of property data from object data, not updates from explicit value lists. Neither **BDB** nor **CBS** I/O modules currently call the **Tcl** or **Python** callback procedures except in table-based processing mode.

The command returns the number of updated records.

Example:

```
molfile rewrite $fh current E_NAME "Black tar, grade A"
molfile rewrite $fh all E_XLOPG2
molfile rewrite $fh [list $mytable records] [list E_IDENT E_REGID]
```

The first command changes the property field `E_NAME` in the current record to the specified value. The second variant recomputes all `E_XLOGP2` values in the file from the stored structure data - for example after updating the computation function of that property, or having added it as a new field to the file. The final version changes the fields `E_IDENT` and `E_REGID` for the records stored in table column *records*, replacing them with the data found in the table columns of the same name.

A complication in the use of this command is that database-type files like the **CACTVS CBS** and **BDB** formats store property definitions themselves. After opening the file, a newly set up property definition, which may for example possess an upgraded computation function, can have been replaced by the old definition from the file. In that case, the new property definition must be explicitly re-read to gain the upper hand again, for example with a **`prop read`** command.

molfile scan

```
molfile scan filehandle|remotehandle expression/queryhandle ?mode?
?parameterdict?
f.scan(query=,?mode=?,?parameters=?)
Molfile.Scan(filename,query=,?mode=?,?parameters=?)
```

Execute a query on the file and return results. The structure file is scanned, by default starting from its current read position, and results are gathered until either the end of the file has been reached (or the scan wrapped once around the file, if the *wraparound* file flag has been set) or a scan condition caused the stopping of the scan procedure. If the scan finished without reaching the end of the file, it can be resumed with another **`molfile scan`** command at a later time.

The file scan works in principle on any file, but with very different efficiency. Files managed by file format I/O modules which support direct field access, and can supply structure and reaction data in binary form, can be queried much (often a factor of 1000 or more) faster than, for example, a plain **SD** file. In the latter format, every record needs to be fully parsed, the structure compared against the query expression, and most of the structure data is discarded immediately after the record has been checked. Files in formats which support various types of indexing for numerical values, bit-screen filtering for super- and substructure searches, hash codes for full-structure matching and other means of acceleration can be effectively queried with typical expressions in a few seconds, even while containing millions of compounds.

The two basic built-in **CACTVS** formats for effective searching are **CBS** (static files, good performance on **CDROM** and other linear media) and **BDB** (efficiently updateable, and with more advanced indexing than **CBS**). In contrast the systematic reading of a million-record SD file takes a few hours. Nevertheless, the feature of universal query support is very useful for working with typical data sets of a few thousand records. These do not need to be converted from their original formats to a query file for a quick exploratory data scan.

Query expression syntax classes

The toolkit currently supports two syntactically unrelated classes of query expressions: Native Cactvs expressions, which are described below, and Bruns/Watson structure queries as described in J. Med. Chem. 2012, 55, 9763-9772, The exact syntax supported is that of the internal Lilly suite in October 2014, which is significantly extended from the description in the paper, but also discards some outdated syntactic elements briefly mentioned in the paper.

Example:

```
set demerits [molfile scan $fh [read_file 9_aminoacridine.qry] {record demerit}]
```

This expression returns a nested list of records which match the query, and their merit/demerit score computed by that rule. Note that records which do not match the expression are omitted, they do not report a zero demerit in the result. Internally, Bruns/Watson queries are mapped to the standard toolkit query expression data structure. Many of the queries in the standard Lilly rule set can be expressed equivalently as a native query. However, at this time there are a few specific Lilly query features which cannot be expressed in native toolkit syntax.

If a query expression cannot be parsed as Bruns/Watson code, an attempt is made to interpret is as native Cactvs expression, and all error messages relate to that interpretation attempt. The following paragraphs all apply exclusively to the native toolkit expression style.

Branch node expression classes

The expression argument is a tree of individual query statements. It is formatted as a nested Tcl list. The he allowed depth of branching as well as the allowed number of leaf nodes is unlimited. The following branch operations are supported in this tree:

- *and*
  One to any number of child branches. The branch query only succeeds if all branches match.
- *or*
  One to any number of child branches. The branch query succeeds if any of the branches match. As soon as the first branch is a match, the other child branches are no longer executed. This is usually desired because it accelerates the processing of the query. However, in some circumstances, for example when computing similarity scores or coloring matched atoms or bonds, this is not the desired behavior. The *orcontinue* operator has the same query branch logic, but all branches are visited.
- *orcontinue*
  See above, an *or* operator variant where all child branches are always executed. This can also be written as *orcont*.
- *xor*
  One to any number of child branches. The branch query succeeds if an odd number of the child branches match. *eor* is an alias name of the operator.
- *not*
  Exactly one child branch. This operator inverts the *match/nomatch* status of the child branch, and lets all other status conditions reported by the child branch pass unchanged.
- *bind objclass*
  One or an odd number of child branches. This is a rather unique operator. Its effect is to force the use of the same minor object in all controlled branches. For example, if the child branches were to contain two molecule property checks connected by an *and* operator, by default the molecules of database structure ensembles which pass these conditions are independent and can be different. If a bind node is located upstream, those two molecules must be the same. Only when the first of a series of conditions is checked, all molecules are iterated as potential matches. If the query continues with a match of the first condition, the molecule is no longer unbound, and only the molecule already matched with the first condition is tested with the other conditions. Bind nodes can be used with any ensemble minor object class on structure queries (such as *atom*, *mol*, *ring*) or ensembles (*ens*) on reaction queries. The *objclass* argument part must be set to the desired class name. Bind nodes only affect controlled nodes which are property queries with properties belonging to the bound object class.

If more than one branch is specified, the query expression branches (first, third, etc. argument) are linked by an identifier which determines how these branches interact under the umbrella of the bind node. The link argument it itself a list. Its first element is the link type identifier (currently one of *independent*, *singlebond* or *doublebond*). Except in case of the first mode, the next element is the index (starting with 0) of the query branch in the bind node. It must refer to an existing branch index, i.e. forward declarations are not possible. For the determination of the branch index only the query branches count. The interspersed link arguments do not generated query branches.

If the mode is not *independent*, the allowed atoms or other minor objects which are tested in the additional branches depend on the current minor object in the referred branch. In modes *singlebond* and *doublebond*, these can only be atoms linked via the specified bond type to the referrer object, not the full atom set of the tested ensemble. In case of linked query branches, these are recursively checked. If a minor object in the leading branch matches, but fails to match in a dependent linked branch, more allowed minor object combinations are tested until they are exhausted or a combination of suitable minor objects is found which matches all branches. In any case, a minor object is only utilized once per bind node, so that for example a chain of three *singlebond* connected query branches needs to match three different atoms - the third branch cannot go back on the bond between the atoms selected for the first and second branch matches.

Example:

```
set q {
bind atom {and {A_ELEMENT in {7 8 16}} {A_NEIGHBORS = 2} {A_RING_COUNT = 0}}
{singlebond 0}
{and {A_ELEMENT = 6} {A_UNSATURATION = 0} {A_RING_COUNT = 0}}
{singlebond 1}
{and {A_ELEMENT in {7 8 16}} {A_NEIGHBORS = 2} {A_RING_COUNT = 0}}
}
molfile scan $fh $q
```

This query tests for a fragment of three atoms, which are connected by single bonds and where the individual atoms are each subject to a check on different set of atomic attribute conditions. The same query could also be realized as a **SMARTS** pattern. The advantage of this notation is that arbitrary properties can be used as attributes and an extended operator set and the full set of comparison mode flags is available. The disadvantage is a less readable pattern representation, and that no substructure query accelerator techniques such as bitvector screening are automatically employed.

- *passswitch*
  A switch where a single child node depending on the current value of the pass index is selected. All other child nodes are ignored in that query pass. This is internally used for *smart similarity* queries and of limited usefulness for normal user-written queries, but it may be used in expert queries. In standard queries, only a single pass, with index zero, is ever executed. The maximum number of passes of a query is determined by the largest number of child nodes in any *passswitch* node in the query.

- *range*
  A node which requires that the number of its children matches are within a range. The syntax of this node is the node type keyword, followed by an (possibly open) integer range indicating the minimum and maximum number of child matches, followed by any number of child nodes of arbitrary type. Regardless of the range check, all child nodes are tested because the *score* pseudo-property can be used in the retrieval set to get the actual number of branch hits.

  Example:

  ```
  range {0-1} [list structure >= $ss1] [list structure >= $ss2] \
  [list structure >= $ss3]
  ```

  This expression requires that zero or one of the three test substructures match.

Here are a few simple expression patterns:

```
molfile scan $fh $leafexpression1
molfile scan $fh [list "and" $l1 $l2]
molfile scan $fh [list "or" $l1 [list "and" $l2 $l3 $l4]]
molfile scan $fh [list "orcontinue" [list not $l1] [list "xor" $l2 $l3]]
molfile scan $fh [list bind mol [list and $l1 $l2]]
```

All branch nodes need to end in leaf expression nodes. An empty query expression is valid and matches every input record. Also, it is legal and actually a common case to have an expression which is just a single leaf node expression. The order of the branches does not matter. An automatically invoked optimizer sorts the branches, and simplify them, in order to achieve maximum performance.

Leaf node expression classes

These are the supported classes of leaf node expressions:

- *all* (or *true*)
  This is just a placeholder. It will matches every record.
- *false*
  This node never matches.
- *filename*
  A condition on the name of the current physical file. This is only useful for scans involving virtual files.
- *formula*
  A molecular formula expression.
- *isnull*
  Check whether property data is absent.
- *notnull*
  Check whether property data is present.

- *property*
  A condition of a property value. If possible, this is evaluated without reading a full structure or reaction object from the file. However, if necessary, the checked property data is extracted from, or even computed on, the full record data item. The first word of a property leaf node expression is the name of the property, not the class name.
- *reaction*
  A reaction query to find records with reactions containing specific bond transformations.
- *record*
  A condition on the file record of the current physical file. For simple single-file scans, this is the same as the virtual record.
- *smartsearch*
  A special variant on the structure search node. This node is internally expanded into four internal alternative queries controlled by a pass-dependent switch node. The expanded queries are a full structure query, a substructure query, and Tanimoto similarity queries with thresholds of 95% and 90%. The complete query is automatically re-run with the next branch of the series of alternative queries until at least one hit has been found. This query mode only works on data sources where the file or other input source can be repositioned to the original start position if a second or later pass is required.
- *structure*
  A structure match operation on the primary database structure, a derived version thereof, or a reaction component. This type of query supports a variety of full-structure, substructure, superstructure and similarity matching methods. Some of these expressions, such as full-structure queries, are internally rewritten to property queries. For full-structure queries, these are hash code checks. Others, such as substructure matching, are handled by special routines. The first word of the leaf node specification can either be *structure*, for the main record structure, which is expected to be cleaned up and standardized, or any other of the recognized structure file ensemble classes (*reagent, product, solvent, catalyst, parent, scaffold, original, deprotected, salt*). If a tested file record does not contain the requested structure variant, an attempt is made to derive it from the main record structure. This works with, for example, the *parent* structure, but not, for example, for obvious reasons with the *original*.
- *true*
  An alias for *all*.
- *vrecord*
  A condition on the virtual file record. For simple files, this is the same as the physical record.

The various leaf expression classes have different syntax schemes, which are explained in the next paragraphs.

record and *vrecord* expressions

The *record* and *vrecord* expression classes are always written with three list elements: The expression class name, the operator, and the value or value list. The operators can be from the standard six numerical types, the range operator (<->), and the *in* or *notin* set operators. Numerical comparisons require a single comparison value, the range operator a pair of values, and the set operators a list. Examples:

```
"record <= 100"
"vrecord <-> {1 1000}"
"record in {1 7 19 230}"
```

filename expressions

The *filename* expression class is even simpler. It always consists of three elements: The expression class name, the operator (which can only be = or !=), and the file name. The actual file comparison operation uses device and inode identifiers on Linux/Unix platforms if the file is accessible, so the exact spelling of any path components does not matter. Example:

```
"filename = part1.sdf"
```

isnull and *notnull* expressions

The *isnull* and *notnull* expression classes are written with two elements. The first element is the expression class name, and the second a property name. The property name may be qualified with an ensemble class modifier. If the modifier is not specified, the query applies to the main database structure. Otherwise, the property of the specified ensemble class is addressed. Examples:

```
"isnull E_NAME"
"notnull product:E_ASSAY_RESULT"
```

random/*subset* expressions

random or *subset* node expression classes (these names are aliases) are written with two elements. The first element is the expression class name and the second a floating point value between zero and one. When this node is encountered for evaluation, a random number between zero and one is generated. If it is less than or equal to the specified value, the node is considered to match. Example:

```
„subset 0.6"
```

This expression will match 60% of the time and let the query proceed for further evaluation or result output.

property expressions

The *property* query expression class is a little bit more complex. It has a variable number of elements, between three and eight. The general syntax scheme is

```
property {operator ?modifiers?..} value ?threshold? ?multimode? ?filter? ?c1? ?c2?
```

The first three elements are always the property name, which can be qualified with an ensemble class, the comparison operator, and one or more values. The number of required values is dependent on the operator. The comparison operator can be a nested list. It needs to contain as a list element the basic comparison operator (numerical, range or *in/notin* set operators) and may additionally contain modifier words, which are translated into flags potentially influencing the datatype-specific comparison functions. It depends on the data type of the property whether any flag word has an effect.

If the *object* flag word is supplied as part of the operator list, the value part of the query is parsed as a chemistry object handle, more specifically an ensemble handle, a decodable string representation of an ensemble, a reaction handle, or a decodable string representation of a reaction. The ensemble variants are accepted if the query property is attached to an ensemble or an ensemble minor object, and the reaction variants can be used if the property is reaction-related. The value of the query is then automatically extracted, even computed if needed, from the object. Properties with fields can be entered with the basic name, or any qualified field name. In addition, the property name may be prefixed by a structure class designator (see paragraph on structure queries). By default a property is assumed to be data of the main structure of the file record, or the main reaction. Examples:

```
"E_NAME = methane"
"solvent:E_NAME {in ignorecase} [list benzene toluene ethylbenzene]"
"E_IRSPECTRUM(source) {= shell nocase} *bruker*"
"E_WEIGHT {<= object} $ehtest"
"E_CAS {= ignoredashes ignorecase} 88337-96-6"
```

These are the comparison flag words which are recognized:

- *absolute*
  Use absolute numerical values for comparison.
- *alternative*
  Use alternative variant of comparison algorithm, if supported. For example, the *bitset/bitunset* comparison methods by default report 0 (equality) only if all bits are identical. The alternative version reports 0 when there is any common bit.
- *approximate*
  Use an approximate version of the comparison operator. For strings, this means that case, whitespace, numbers and punctuation are ignored. For floating point data, it means that the comparison employs rounded integer values. This can also be specified by an *at* @ character directly attached to the operator.
- *asnumber*
  Extract number from, for example, a string and use that for numerical comparison instead of literal comparison.
- *bitset*
  Interpret the query expression value as bit mask and check whether all bits in that mask are also set in the file value.
- *bitunset*
  Interpret the query expression value as bit mask and check whether all bits in that mask are unset in the file value.
- *contained*
  Test whether the query expression value is contained in the file value. For strings, this is simple substring matching. For vectors, this is an element match.
- *cosine*
  Compute cosine similarity coefficient percentage from query expression and file value and remember this as score. This comparison is only supported for bit vectors.
- *correlation*
  Compute correlation coefficient from numerical vector types.
- *dice*
  compute Dice similarity coefficient on bit vectors, bit sets or strings (via bigraphs).
- *euclidean*
  Compute Euclidean distance from numerical vector types.

- *extended*
  Use an extended version of a comparison method. For example, in conjunction with regular expressions, this enables extended regexp syntax.
- *glob*
  Interpret query value as shell expression. This can also be specified by an asterisk * character directly attached to the operator.
- *ignorecase*
  Ignore case for string-related comparisons. This can also be specified by an *i* character directly attached to the operator.
- *ignoredashes*
  Ignore dash/minus characters in string-related comparisons
- *ignorewhitespace*
  Ignore whitespace in string-related comparisons
- *ignorezero*
  For numerical vector comparisons, ignore zero elements.
- *needelementmatch*
  For vector comparisons with the *contained* flag, the default method is to check whether all elements of the query vector value compare to one element in the file vector data, but not necessarily in the same position. If this flag is supplied additionally, any single element match will suffice for a positive comparison result.
- *needelementmismatch*
  For vector comparisons with the *contained* flag, the default method is to check whether all elements of the query vector value compare to one element in the file vector data, but not necessarily in the same position. If this flag is supplied additionally, there needs to be at least one element mismatch for a positive comparison result.
- *object*
  Decode value as object, and compute comparison value from it The the object is a string representation, the object is only created temporarily and discarded as soon as the value was obtained. Persistent objects that are addressed via their handles remain valid and unchanged, except that their property data set is potentially extended by the computation.
- *precision*
  Use the precision as defined in the property description to check for equality. By default, full CPU precision is used.
- *regexp*
  Interpret query value as regular expression. This can also be specified by a tilde ~ character directly attached to the operator. Starting with toolkit version 3.352, the regular expression syntax on all platforms is that of the PCRE library, also known as the Perl style.
- *swap*
  Swap left and right side of the expression in the comparison. This makes especially sense for asymmetric operations such as regular or shell expressions. With a *swap* word, the regular or shell expression is the string from the file, not the written query value.

- *tanimoto*
  Compute Tanimoto similarity coefficient percentage from query expression and file value and remember this as score. This comparison is only supported for bitsets and bit vectors.
- *tversky*
  Compute Tversky similarity coefficient percentage from query expression and file value and remember this as score. This comparison is only supported for bitsets and bit vectors.
- *trim*
  Ignore leading and trailing whitespace. Spaces in the middle of a string are still significant.
- *unique*
  Hint for the query processor that the value is expected to match only once in the file, if at all. This is useful for query optimization. If a hit has been found, additional records need not to be checked.
- *vectorrange*
  For numerical vector comparisons. The query expression value vector is expected to contain twice as many elements as the file values. Every pair of values in the query vector is interpreted as a required upper and lower bound for the file values.
- *withdigits*
  In conjunction with the *approximate* modifier, make digits significant again.

If the operator is the *in* or *notin* word, the value part is interpreted as a list. The value, or value list item, must be parseable according to the property data definition. Enumerated values and similar encodings may be used if properly defined in the property descriptor record.

If the comparison function computes a score (for example, the Tversky or Tanimoto variants), the next optional argument is a threshold value which needs to be exceeded to register as hit. If the threshold parameter is not specified, or given as a negative value, any score passes. Example:

```
"E_SCREEN {>= tanimoto object} $eh 95"
```

The next two optional arguments concern the case when there is more than one file data value to compare against the expression value. This generally happens when the tested property is not a major object property, but a minor object property, such as an atom or molecule property. In that case, the database record often contains multiple values, because there is more than one atom, or more than one molecule in the structure in the record. The first argument is the general match criterion. It can be set to *one*, *all*, *none*, or *both*. The default is *one*. Mode *one* means that it is sufficient if one of the record values matches. Mode *all* requires all to match, mode *none* requires that none matches, and mode *both* requires that there are both matches and mismatches.

The next optional parameter is a filter which can be used to restrict the values tested. If it is not present, or an empty string, no filter is applied. Example:

```
"A_ELEMENT = 6 {} all ringatom"
```

Above expression checks whether all ring atoms in the structure are carbon. Any record with a hetero ring atom fails the test.

The final two optional arguments are integer constants which may be used by the comparison operation. If they are not specified, both are implicitly passed as zero. If the first is specified, but not the second, the second is set to 100 minus the first value. Almost all comparison operations on the various data types ignore these.

One comparison mode which does make use of them is the Tversky bit vector similarity score. Here *c1* and *c2* are the weights of the bits in the first and second compared value. For scoring, both parameters are divided by one hundred and the floating point results are used as weight multipliers. Example:

```
"E_SCREEN {>= tversky object} $eh 90 {} {} 30 70"
```

Above expression computes a Tversky score on the standard structure search screen `E_SCREEN` with 30% weight for the database structure features and 70% of the query structure features (i.e. imbalanced towards a substructure rating), and report the record if the score is 90% or higher.

Starting with version 3.358 of the toolkit, property expressions where the data type of the query property is *structure* or *reaction* are no longer parsed as standard property expression, but as structure or reaction query expressions, respectively. Example:

```
"V_ONTOLOGY_TERM(substructure) {>= swap  stereo isotope charge} $eh"
```

Since the data type of the field of `V_ONTOLOGY_TERM` is *structure*, the syntax rules of normal property expressions no longer apply. Instead, the syntax for structure expressions explained below is substituted.

structure expressions

Structure expressions are used to invoke structure comparison operations, such as sub- and superstructure search. The expression is a list, with three to eight elements. A structure expression starts with the structure identifier, followed by the operator, which, as in property queries, may be written as a list with auxiliary modifier words, and as third mandatory argument the comparison structure source.

The structure identifier is the name of a structure class. Usually it is present as part of the record in the queried file, but some structure classes can be computed from the main structure if necessary. If a structure class can neither be found in a file record, nor computed, the node will not match. The following structure classes are supported:

- *structure*
  The main structure. Usually expected to be a standardized, normalized form.
- *original*
  An original structure, un-standardized. *deposited* is an alternative name.
- *salt*
  A salt form
- *deprotected*
  A variant without protective groups
- *parent*
  A parent compound. There is a standard computation function for this form.
- *scaffold*
  A structure core, isolated by some algorithm.
- *reagent*
  A reagent ensemble. Usually this is a part of a reaction record, but it can be present also on its own.

- *product*
  A product ensemble. Usually this is a part of a reaction record, but it can be present also on its own.
- *solvent*
  Solvent for a reaction. Usually this is a part of a reaction record, but it can be present also on its own.
- *catalyst*
  Catalyst for a reaction. Usually this is a part of a reaction record, but it can be present also on its own.

At minimum, the operator section (the second, mandatory argument) contains a standard numerical operator symbol. Additionally, modifier words may be present as additional list elements. The following operators are supported.

- =
  Structure identity, i.e. full-structure search. This is internally re-written to an equivalent hash code search as a property comparison node. A suitable hash code is automatically selected depending on the operator modifiers such as *stereo* and *isotope*.
- !=
  Structure inequality, i.e. a negated full-structure search. This is internally re-written to an equivalent hash code search as a property comparison node. A suitable hash code is automatically selected depending on the operator modifiers such as *stereo* and *isotope*.
- >=
  Substructure search.
- >
  Substructure search, excluding identity.
- <=
  Superstructure search. This operation ignores hydrogens on the database structures (see below).
- <
  Superstructure search, excluding identity. Superstructure search ignores hydrogens on the database structures when the database entries are used as sub-graphs - otherwise a normal, fully specified database molecule will not match much. For the identity check, hydrogens are significant.
- ~>= or ~>
  Tanimoto similarity search with a reporting limit. This is internally re-written to an equivalent property search.
- %>= or %>
  Tversky similarity search with a reporting limit. This is internally re-written to an equivalent property search.

- <->
  Substructure match count range search. This automatically changes the default substructure match mode to *distinctinneratoms* (see **`match ss`** command and the *count* modifier below). The optional fourth argument can be used to set a range condition for this mode. If only a single number is supplied, the match count for a successful node match must be exactly this number. If a list of two numbers is used, these define a range of acceptable match counts. If no explicit range is set, its implied value is one to 65535. It is possible to use a lower bound of zero which lets structure mismatches pass the query condition. This can be useful when match-dependent data is retrieved, for example the *matchcounts* pseudo property (see below).

The default substructure match mode has the *bondorder*, *useatomtree* and *usebondtree* flags set (see **`match ss`** command). The initial flag set can be modified with modifier words linked to the operator. As far as it makes sense, the modifier words also change the operation of derived query modes, such as full-structure matching via hash codes.

These are the modifier words which can be used in structure expressions:

- *absolutestereo*
  Perform absolute stereo matching. By default, stereochemistry is not used in the query, except if set up explicitly as atom- or bond-specific query attribute in properties `A_QUERY` and `B_QUERY` as part of the query substructure specification. An alternative syntax is to directly attach an uppercase *S* character to the operator.

- *allowmissingstereo*
  If set, absent stereochemistry descriptors in file structures can be matched by explicit stereo centers in the query structure. However, stereo center mismatches still lead to a match failure.

- *anyfragment*
  Report a match for full-structure search if any molecule of the file structure is identical to the query structure. For substructure/superstructure queries, this flag has no effect, since their default operation mode already covers the effects of the flag.

- *anyoverlap*
  If the substructure contains multiple fragments, they may match overlapping parts of the structure ensembles. By default, matched substructure fragments cannot overlap. This flag cannot be combined with *atomoverlap*.

- *arotautomer*
  A more aggressive form of the *tautomer* mode. In this mode, tautomers involving the dissolution of aromatic systems are also found, in addition to the more low-energy tautomer forms matched with the normal *tautomer* mode.

- *atomoverlap*
  If the substructure contains multiple fragments, they may match overlapping atoms, but not overlapping bonds. By default, matched substructure fragments cannot overlap at all. This flag cannot be combined with *anyoverlap*.

- *charge*
  Match formal charges of query atoms. By default, charges are not compared, except if set up explicitly as atom-specific query attribute in property `A_QUERY` in the query substructure specification.

- *count*
  For substructure and superstructure matching, check not only for the presence of a match, but count the number of distinct matches equivalent to the match mode *distrinctinneratoms* in the **`match ss`** command. The normal substructure match mode is equivalent to the first mode in the **`match ss`** command, yielding only counts zero or one.

- *distinctatoms*
  Set the substructure match mode to this value (see **`match ss`** command). Has an effect only for substructure matches. The default substructure match mode is *first*, except if the match operator is *range* for counted pattern matches. In that case, it is *distinctinneratoms*.

- *distinctfgatoms*
  Set the substructure match mode to this value (see **`match ss`** command). Has an effect only for substructure matches. The default substructure match mode is *first*, except if the match operator is *range* for counted pattern matches. In that case, it is *distinctinneratoms*.

- *distinctheavyatoms*
  Set the substructure match mode to this value (see **`match ss`** command). Has an effect only for substructure matches. The default substructure match mode is *first*, except if the match operator is *range* for counted pattern matches. In that case, it is *distinctinneratoms*.

- *emptyssismismatch*
  By default, a substructure without any atoms matches anything. If this flag is set, it matches nothing instead.

- *exactaro*
  Match aromatic bonds exactly. By default, simple single or double query structure bonds match structure file record aromatic bonds.

- *exactringsystem*
  Rings in substructure fragments must match complete ring systems only. For example, with this flag a benzene substructure no longer matches naphthalene, anthracene, etc. Non-ring parts of the substructure can still, if other query attributes do not prevent this, match both ring and chain parts of file structures. For full-structure queries, this flag has no effect.

- *extended*
  Use extended versions of the match procedures. For similarity queries, this enables the **PubChem** extended scoring mechanism. If the query structure is identical to a file structure both in stereochemistry and isotope labels, an artificial score of 104 is computed, 103 if isotopes or stereochemistry match, but only one of these, 102 for basic equivalence of connectivity without isotopes or stereochemistry, and 101 for a tautomer. Compounds which are not structurally identical to the query structures using one of these criteria are scored normally.

- *fragmentsplit*
  Treat every molecule in the query structure as a separate fragment. The query ensemble is implicitly split, and every component therein is stored in an independent structure expression node. These nodes are then connected with an *or* or *orcontinue* branch mode. This is similar to using a file handle pointing to a file with multiple records as query structure data source (see below).
- *framework*
  Substructure carbon atoms cannot have any unmatched, directly bonded carbon or hetero atom neighbors in the structure. Unmatched bonded hydrogen is allowed. This flag has an effect only for sub- and superstructure match modes.
- *implicitsinglearo*
  If this flag is set, bonds which were created with an implicit bond order when the query structure was decoded are matched as if they were explicit *single/aro* query bonds. This is a useful mode for emulating Daylight software.
- *isotope*
  Perform isotope matching. By default, isotope labels are not used in the queries, except if set up explicitly as atom-specific query attribute in property `A_QUERY` in the query structure specification. An alternative syntax is to directly attach an *i* character to the operator.
- *matchallheavyatoms*
  Require that all heavy atoms in the file structures are matched. This feature generates matches of file structures similar to full-structure matches while allowing the use of substructures with variable match conditions, such as atom lists.
- *nobondorder*
  Do not compare bond orders. This flag has an effect only for sub- and superstructure match modes.
- *nochainonaro*
  Do not match chain parts of the query substructure on aromatic bonds in the file structures. This flag has an effect only for sub- and superstructure match modes.
- *nochainonring*
  Do not match chain parts of the query substructure on ring bonds in the file structures. This flag has an effect only for sub- and superstructure match modes.
- *nodoubleonaro*
  Do not match otherwise unmarked double bonds in the substructure onto aromatic bonds of the structures.
- *noquerytree*
  Deactivate extended matches requiring full checks of the query tree fields in the `A_QUERY` and `B_QUERY` properties in the query structures. Certain query inputs need these trees for precise matching, because the query cannot be expressed as a flat set of query attributes. Examples for queries requiring tree matching for proper execution are complex SMARTS expressions beyond those using only simple explicit or implicit *and* in atomic or bond expressions, and Recursive SMARTS. Disabling the flag may lead to a small speed-up for simple substructure queries.

- *nosingleonaro*
  Do not match otherwise unmarked single bonds in the substructure onto aromatic bonds of the structures.

- *nosubstructureh*
  For substructure match, ignore any hydrogens present in the query structure. This is a convenient shortcut to allow the use of hydrogen-complete structures as simple substructures. A similar scheme is automatically invoked for superstructure search, where hydrogens in the file structures are ignored in matching.

- *reactionflags*
  Match reaction transform flags in the substructure. Both query and file structures need to have data for property `B_REACTION_CENTER` set. The supported set of comparisons is compatible with **MDL's** ISIS database. Note that this flag can be used gainfully in structure expressions for half-reaction matching. It is not limited to full reaction queries. This flag is on by default in reaction queries, but off for structure queries.

- *relativestereo*
  Perform relative stereo matching. By default, stereochemistry is not used in the query, except if set up explicitly as atom- or bond-specific query attribute in properties `A_QUERY` and `B_QUERY` in the query structure specification. An alternative syntax is to directly attach a lowercase *s* character to the operator.

- *sethighlight*
  In case structure ensembles are retrieved from the file (**`molfile scan`** modes *ens*, *enslist*, *reaction* or *reactionlist*), the bonds and atoms matched by a substructure are marked in the returned structure-side ensembles with the highlight flags in properties `B_FLAGS` and `A_FLAGS`. In case multiple matches occur, the highlight set is an union of all processed matching substructure mapping. This flag is also automatically set if the property retrieval set in the **`molfile scan`** command includes related pseudo properties, such as *matchatoms* or *matchbonds*.

- *setmatchproperty*
  In case structure ensembles are retrieved from the file (**`molfile scan`** modes *ens, enslist, reaction or reactionlist*), the bonds and atoms matched by a substructure are marked in the returned structure-side ensembles by attached properties `A_SSMATCH` and `B_SSMATCH`. These are set to the labels of the matching substructure atoms or bonds. Unmatched structure ensemble parts have match property values of zero. In contrast to the *sethighlight* flag, this option attaches a new match property instance for any successful and processed match. Returned ensembles may therefore possess series of property instances like `A_SSMATCH`, `A_SSMATCH/2`... and so on.

- *swap*
  Swap the left and right structures in the query. This means, for example, that the database is expected to contain substructure definitions, and the query value argument a fully defined structure. This is not exactly the same as a superstructure search because of the different style how hydrogens are handled. For superstructure search, hydrogen atoms in the file records are ignored, generating a simplified structure from the record data for matching, but in case of a swapped substructure search, the file record is submitted as substructure for matching without any processing.

- *tautomer*
  Match tautomers of the query structure. If this flag is active, non-aromatic single and double bonds in tautomer systems need not to be matched exactly, as long as the overall bond order count is a match. Mobile hydrogens can either be specified explicitly, or a full implicit set can be used if the *useimplicith* flag of property `B_ISTAUTOMERIC` is set. The standard mode does not consider tautomeric forms which destroy aromatic systems. If you need to find matches between aromatic and non.aromatic tautomer systems, use the more aggressive *arotautomer* mode.

- *unique*
  Hint for the query processor that the query ensemble is expected to be matched only once in the file, if at all. This is useful for query optimization. If a hit has been found, additional records need not to be checked.

Many of these global flags can be overridden, or activated on a local level, for individual atoms or bonds, in the `A_QUERY` and `B_QUERY` properties. For example, `A_QUERY` has fields for flags which can request the matching of stereo or charges for specific atoms, or to allow missing stereochemistry at a specific center. These per-atom or per-bond requests override global query flag settings.

The third mandatory expression list element is the structure source. It can be one of

- an ensemble handle
  The ensemble is directly decoded.

- a list of ensemble handle and molecule label
  The fragment indicated by the molecule label is extracted from the ensemble and used for the query as isolated entity. If the molecule label cannot be found, an error is reported.

- structure line notation string
  For example, a **SMARTS/SMILES/SLN/INCHI/CID** string or a packed **CACTVS** ensemble - anything which can be decoded by the **`ens create`** command. The string is decoded into a transient ensemble, which is automatically discarded when it is no longer needed. The exact decoding specifications depend on the operator. For full-structure search, a fully specified structure is created, while for substructure-type queries implicit hydrogens are not attached, and the full range of query specifications of the encoding format is allowed.

- a dataset handle or reference
  A dataset containing at least one ensemble. All dataset ensembles are checked, and internally for every ensemble a separate expression node is created. The nodes are then linked via an *or* or *orcontinue* (in case a scoring operator is used) branch node. Dataset objects which are not ensembles are silently ignored. The hydrogen status of the dataset ensembles is not changed. In case there is only a single ensemble in the dataset, this command is indistinguishable from using the ensemble handle directly. In case the dataset does not contain any ensembles, an error is raised.

- a reaction handle or reference
  A reaction containing as least one ensemble. All reaction ensembles are checked, and internally for every ensemble a separate expression node is created. The nodes are then linked via an *or* or *orcontinue* (in case a scoring operator is used) branch node. The hydrogen

status of the reaction ensembles is not changed. In case there is only a single ensemble in the reaction (i.e. a half-reaction), this command is indistinguishable from using the ensemble handle directly. In case the reaction does not contain any ensembles, an error is raised.

- a *molfile* handle or reference
  An opened structure file. All remaining records are read, and internally for every record a separate structure expression node is created. The nodes are then linked via an *or* or *orcontinue* (in case a scoring operator is used) branch node. If the match operation is full-structure, the file is read with automatic hydrogen addition (see **`molfile set`**), otherwise without any conversion flags. However, since the hydrogen addition flag is the only file attribute which may be temporarily overridden, other molfile object attributes may be set before the file is used in the query expression. Of course, using a file with a huge number of records in this fashion may cause problems. In case the file does not contain any records behind the read pointer at the time the command is parsed, an error is raised.

Query specifications found in structure sources are understood in a variety of formats. **DAYLIGHT** and **MDL** formats are decoded and translated into an internal representation in an almost completely compatible fashion. That includes **RECURSIVE SMARTS**, **ISIS 3D** queries, **MDL** stereo groups and **MDL** reaction queries. A significant range of **SYBYL SLN** and **CAMBRIDGESOFT CHEMFINDER** query expressions are also understood, as well as features found in the **CSD CONQUEST** software. Finally, in **CACTVS** there is no fundamental difference between a query fragment and a normal structure object. Query structures are just structures with additional information stored in properties `A_QUERY`, `B_QUERY` and possibly `B_REACTION_CENTER`. For basic matching, any structure object will do, even if they do not possess these query attribute properties. However, an eye should be kept in the hydrogen status of query fragments. If no specific flags are set, substructure matches attempt to match hydrogen atoms just like any other atom. Example:

```
set ehss [ens create C]
set ehss [ens create C smarts]
```

The upper substructure ensemble does not, in the absence of hydrogen ignore flags, match any structure ensemble except those which contain a full methane (one C plus four H) molecule as fragment, because that is what the substructure represents. The second code line decodes the substructure in full **SMARTS** mode. Not only now the full range of **SMARTS** expressions can be parsed (though absent in this example), but the structure is also be created without implicit hydrogens. The first substructure could still be used in a **`molfile scan`** command as a simple carbon match test if the *nosubstructureh* modifier flag were supplied.

In order to read query structures from a file, the following generic open statement is the standard approach:

```
molfile open $file r hydrogens asis readflags noimplicith
```

Simple query formats, such as **MDL ISIS** query *Molfiles*, are read into a flat set of attributes. More complex formats, such as **SMARTS**, may require the use of a tree of expressions on individual atoms and bonds, similar to the overall query tree with branch and leaf nodes described here for the **`molfile scan`** command. These complex formats are nevertheless also translated, to the degree possible, to the flat model. For example, a **SMARTS** expression with only uses simple atom lists or atom and bond query attributes all connected just by *and* can be fully represented in this way. This also means that, format translation into other query file formats is also possible for these simple expressions. The use of the full query trees in matching can in some cases be a performance issue.

The *noquerytree* flag is available to restrict the match to those parts of the full query which can be expressed in the flat model.

The fourth and optional expression list element in the query expression is used only for a few match modes. If it is not set, the default value is minus one.

• similarity queries:	The minimum score required to report a hit
• substructure count ranges:	A list of the acceptable minimum and maximum occurrence counts of the substructure. If only a single value is supplied, is is used both as minimum and maximum value. If not set, the implicit range is 1 to 65535.

Example:

```
"structure ~=> $eh 90"
"product <-> C(=O)\[OH\] {2 3}"
```

The first sample expression is a standard Tanimoto similarity query, with a 90% threshold. The second query matches product structures with two to three carboxyl groups.

Optional expression list elements five and six correspond to the *c1* and *c2* parameters in property query expressions. These are currently only used in Tversky similarity queries:

```
"structure %>= $eh 90 30 70"
```

This is an expression for a skewed Tversky similarity (70% query structure, 30% file structure weight) with a 90% reporting threshold.

The seventh optional structure expression list element can be used to specify exclusion substructures. It only applies to substructure matching. In this mode, the parameter encodes a list of substructures which are matched first on the test structure, before the actual substructure match. All atoms which are matched by the exclusion substructures are blocked from consideration in the main match operation. Every exclusion list element can either be an ensemble handle, a list consisting of an ensemble handle and a molecule label, or a structure line notation string (usually a **SMARTS** string) which is decoded in default pattern mode. Exclusion substructures are for example useful to hide structure parts which are already matched by a different pattern, without actually removing structure atoms. Exclusion substructures are always matched exhaustively, so a single exclusion fragment can block multiple matched structure locations.

An example:

```
set ss [ens create {C=C=C.C=C} smarts]
set q "and {structure {<-> exactaro distinctfgatoms} {$ss 1} 1} {not {structure
{>= exactaro} {$ss 2} {} {} {} {{$ss 1}}}}"
echo [dataset scan [list C=C C=C=C C=C=CCCCC=C] $q reclist] (2)
```

The example scan only matches the second test structure. It first tests that the first (allene) fragment is matched exactly once (under application of the *distinctfgatoms* duplicate filter, so two different possible positionings of the substructure on the structure count only once) by the test structure, and then checks that the second (ethylene) fragment does not match the sane structure. Without an exclusion substructure on the second substructure match node, the test would always fail because the ethylene fragment also matches part of the larger allene fragment. In order to prevent this, the negative ethylene query also uses the allene fragment as exclusion fragment. In that case, all carbons in the second test structure are covered, and the query succeeds. In the third test structure, the allene exclusion fragment also covers part of the test structure, but the true simple ethylene part remains unblocked and matches the negative structure query, which results in overall rejection.

An optional eighth argument can be used to fine-tune how exclusion matches are processed. It can be a bitset combination of the enumerated values *burnatoms*, *burncarbon*, *burnterminals*, *burnringsystems* and *burnaroringsystems*. The default burn mode is *burnatoms*. In any case, the exclusion processing only applies node-locally - every node is independent. Exclusion marking does not apply to the matching of other exclusion fragments in the node in case more than one fragment is tested, so these may overlap in their matched structure parts.

The effects are:

- *burnatoms*:
  All structure atoms matches by the exclusion fragments can no longer be matched by the main match.
- *burncarbon*:
  All structure carbon atoms matches by the exclusion fragments can no longer be matched by the main match. Hydrogen or hetero atoms matched by exclusion fragments are still matchable.
- *burnterminals*:
  All terminal structure atoms (those with less than two bonds) matched by the exclusion fragments are excluded from future matching.
- *burnringsystems*:
  All structure atoms which are in a ring and are matched by the exclusion fragments are marked. In addition, any other atom in structure ringsystems where one or more of the ring atoms has been matched are also marked. The excluded atom set is thus usually larger than the test fragment.
- *burnaroringsystems*:
  All structure atoms which are in a aromatic ring and are matched by the exclusion fragments are marked. In addition, any other atom in aromatic structure ringsystems where one or more of the ring atoms has been matched are also marked. The excluded atom set is thus usually larger than the test fragment. In case a ring system consists of both aromatic and aliphatic rings, only the atoms of the aromatic rings are marked.

Example:

```
set q {structure >= [c][OH] {} {} {} {{[n]}} burnaroringsystems}
echo [dataset scan {c1ccccc1O c1cccnc1O} $q reclist]
```

Above query for a phenolic substructure only matches the phenol (first) molecule. The hydroxypyridine (second) molecule is excluded because the exclusion fragment (aromatic nitrogen) not just blocks the nitrogen with the nonstandard burn mode, but the whole aromatic ring it is part of so the aromatic carbon in the main test structure can no longer match. If a test structure had both non-annealed phenol and hydroxypyridine moieties, the match would again succeed because only the aromatic carbons of the hydroxypyridine would have been excluded.

If exclusion fragments are used, the test structures must be fully expanded, i.e. a direct accelerated match on Minimols is not possible.

If the file format supports it, bitvector screening is automatically be applied to reduce the number of records for which structures need to be loaded and sent to graph-based atom-by-atom substructure matching. The default structure match screening property is `E_SCREEN`. The standard versions of `E_SCREEN` implement three predefined fragment sets. The higher sets are identical to the lower ones in the leading bits. Sets zero to two, which yield bit vectors of increasing length and selectivity, but also storage requirements can be requested by setting

```
prop setparam E_SCREEN extended 0/1/2
```

The bit set read from the query file must correspond to the parameter setting for `E_SCREEN` in the current **Tcl** interpreter, if the screen bits are automatically computed on the query structure. The **CBS** and **BDB** file formats, which are optimized for structure query operations, contain screen bit version information in the file header and automatically configure the property parameter setting when the file is opened. For other file formats with screen bits this needs to be done explicitly in the application script. It is also possible to change the structure bit-screen property associated with a file by setting the appropriate *molfile* handle attribute, so it is easily possible to use custom screen bit sets instead of the default property.

Starting with version 3.358 of the toolkit, property query expressions where the data type of the property is *structure* are automatically parsed as structure expressions.

smartsearch expressions

This query expression takes the same arguments as a *structure* expression. It is internally expanded into four alternative queries, linked by a pass-dependent switch control node. The four alternative queries are a full-structure query (equivalent to operator = in a structure query), a substructure query (operator >=), and two Tanimoto similarity queries with thresholds of 95% and 90% (operator ~>=).

When such a query expression is a component of query expression tree, the query is first run with the full-structure query. If that query yields less results than the pass match limit (by default one, i.e. the query does not match anything, this can be configured via the molfile *passlimit* attribute), the input data source is repositioned to the original start record and then the substructure query is run, and if that run also does not yield sufficient hits, the two similarity queries are tried one after another.

Running the second and later alternatives is only possible of the data source can be repositioned to the original start position of the first pass. If that fails, the query is silently terminated early. The pass match limit comparison triggering the possible re-execution of the query is with the global hit count of the query, not the number of hits returned by the *smartquery* branch. If other parts of a complex query produce sufficient hits, the query is not re-run even if a *smartquery* branch did not return any hits.

Hits returned in different passes can be distinguished by including the *pass* pseudo-property in the retrieval data.

By convention, smartsearch expressions are written with an = operator. The actual operator in a *smartsearch* expression is ignored, but modifiers are not. So specifying options like the use of stereochemistry or isotopes is supported and useful.

It is possible to have multiple smart search expressions in a query. The query pass index for these is incremented in parallel, not independently.

The smart search feature was inspired by a similar functionality in the Accelrys *Isentris* system.

Examples:

```
"smartsearch = c1ncccc1"
"smartsearch {= stereo} \"L-lysine\""
```

formula expressions

Formula expressions are used to match file structures by element composition. Conceptionally, this is a special syntax for a complex property match on file structure properties `E_ELEMENT_COUNT` and `M_ELEMENT_COUNT`. A formula search expression is always a list of three elements. The first element is always *formula*, the second element the comparison operator, and the third word the formula specification. The following operators are supported:

- =
  Match the formula specification. There cannot be any elements present in the structure which are not mentioned in the formula.
- >=
  Match the formula specification. Elements which are not mentioned in the formula may be present in the tested structure.
- >
  Match the formula specification. At least one element which is not mentioned in the formula must be present in the tested structure.

For formula queries, there are no modifier words for the operator.

The syntax of the formula is built on the lowest level by element or pseudo-element symbols, which may be grouped into sum or difference expressions and may possess a prefixed count multiplier. The symbol or symbol group can then be suffixed by a simple count, or an open or closed count range. If no count range is specified, the default count is one. In case an element is entered more than once, all counts for that element are added. Finally, the expression may be grouped by period characters into sub-expressions to be applied to different molecular fragments in the tested structures.

Besides normal elements, the following pseudo-elements, which are compatible to the set of the **CSD** *ConQuest* software, are recognized:

- ?
  An atom in the tested structure which is not a simple element.
- [Any]
  Any atom which is a simple element (SLN syntax)
- [Hev]
  Any atom which is a simple element and not hydrogen (SLN syntax)
- [Het]
  Any atom which is a simple element and neither carbon nor hydrogen (SLN syntax)
- [1A]
  Elements from the first PSE main group, excluding hydrogen (Li, Na, ..).
- [2A]
  elements from the second PSE main group (Be, Mg, ..)

- [3A]
  Elements from the third PSE main group (B, Al, ..)
- [4A]
  Elements from the fourth PSE main group (C, Si, ..)
- [5A]
  Elements from the fifth PSE main group (N, P, ..)
- [6A]
- Elements from the sixth PSE main group (O, S, ..)
- [7A] or [Hal]
  Elements from the seventh PSE main group (F, Cl, ..)
- [8A]
  Elements from the eighth PSE main group (He, Ne, ..)
- [1B]
  Elements from the first PSE minor group (Cu, Ag, ..)
- [2B]
  Elements from the second PSE minor group (Zn, Cd, ..)
- [3B]
  Elements from the third PSE minor group (Sc, Y, ..)
- [4B]
  Elements from the fourth PSE minor group (Ti, Zr, ..)
- [5B]
  Elements from the firth PSE minor group (V, Nb, ..)
- [6B]
  Elements from the sixth PSE minor group (Cr, Mo, ..)
- [7B]
  Elements from the seventh PSE minor group (Mn, Tc, ..)
- [8B]
  Elements from the full eighth PSE minor group (Fe, Co, Ni, Ru, Rh, ..)
- [8X]
  Elements from the first column of the eighth PSE minor group (Fe, Ru, ..)
- [8Y]
  Elements from the second column of the eighth PSE minor group (Co, Rh, ..)
- [8Z]
  Elements from the third column of the eighth PSE minor group (Ni, Pd, ..)
- [1M]
  Metals from the first and second main groups (Li, Na, Mg, K, Ca, ..)

- [2M]
  Metals from the third to sixth main groups (Al, Ga, Ge, Sb,..; but not Si, As, Se, Te)
- [3M]
  All main group metals (union of [1M] and [2M])
- [TR]
  ll transitions group metals, no main group elements or lanthanides/actinides
- [LN]
  Lanthanides
- [AN]
  Actinides (no, this is not [AC]!)
- [4M]
  All metals in the PSE
- [NM]
  All non-metallic elements

Element items can be grouped with round brackets into sums or differences. However, this is no full arithmetic expression parser. Element symbols can only be used as stand-alone syntactic elements, bracketed all-sum expressions, or bracketed all-difference expressions.

An element or an arithmetic group can have an appended count. This count can be:

- missing
  The default count is one.
- a simple integer
  The count must be matched exactly.
- a full integer range
  The count must lie between the minimum and maximum values.
- an open range
  Left-open ranges have an implicit minimum count of zero, right-open ranges an implicit maximum count of infinity.
- an asterisk
  This is the same as a right-open range starting with zero, i.e. zero to any number of occurrences.
- a plus character
  This is the same as a right-open range starting with one, i.e. one to any number of occurrences.
- a standard numerical comparison operator, followed by a number
  The value is compared according to the specification. This is a **CSD** compatibility feature.

Examples:

```
"formula = C6H6"
"formula = C5-6H6-"
"formula >= (Cl+Br)2"
```

```
"formula > \[4M\]>=3" or {formula > [4M]>=3}
"formula = (2C-H)-6"
"formula = CH3COOH"
"formula = \[Het\]>1" or {formula = [Het]>1}
"formula = N1-{0.25C}"
```

The first expression is a simple test which matches any ensemble with a composition of six carbon and six hydrogen atoms. The second looks for compounds with five to size carbon and six or more hydrogens, but no other elements. The third example finds compounds where the sum of chlorine and bromine atoms is two. Other elements may be present but are not required, so this expression matches Cl_2, Br_2 and ClBr as well as dichlorobenzene. The fourth expression finds structures with three or more metal atoms. The fifth expression finds compounds where twice the carbon atom count minus the count of hydrogen atoms has a value up to six. Element sum and difference multiplier factors may be floating point numbers, but the ultimate comparison step is performed with the rounded sum or difference by integer comparison. The next line finds compounds with a formula of $C_2H_4O_2$. The counts for elements repeated in the formula string are summed up. The next example matches any compound with one or more hetero atoms. The square brackets in the first writing style are properly escaped to survive standard `Tcl` command parsing

The final example shows how to use computed comparison values, which are specified within curly braces. This expression matches compounds which contain at least one nitrogen, but the number of nitrogens cannot be more than a quarter of the carbon count. For computed comparison values, only natural elements and the `[Hev]`, `[Het]` and `[Any]` pseudo elements are currently recognized. At this time, only a single element, optionally prefixed by a floating-point multiplier and adjusted by a positive or negative floating-point offset, is supported in the specification of a computed comparison value.

Vertical bars can be used to define separate formula match sections. These are applied to individual molecules in the tested structures, not the full ensemble. If a single bar is specified at the beginning or end of the expressions, it signifies a single expression section to be applied to a molecule. When a test for formula sections is applied, all permutations of possible matches between the molecules in an ensemble and the formula expression sections are tried. It is neither required that there is any specific order of the molecules in the ensemble, nor a specific order in the formula expression sections, not is there a need for a match between the molecule and formula section count. However, every expression section in a formula needs to match a different molecule in the tested ensemble for a final match.

Examples:

```
"formula = C6H6|C7H8"
"formula = |H2O"
```

The first expression looks for ensembles which contain one molecule with the formula C_6H_6, and another with formula C_7H_8. The second expression matches ensembles with one or more water molecules. In both cases, molecules/fragment with different composition may be present in the record. In order to test for two or more formulae with the additional conditions that there are no other molecules/fragments, use two formula expression nodes connected with an *and* branch node, as in

```
and "formula = C6H6|C7H8" "formula = C6H6C7H8"
```

Element symbols which stand for specific isotopes, such as *D* for deuterium, are currently not processed. *D* and *T* are read as a simple alias for hydrogen, disregarding the isotope label.

It is possible to use an ensemble handle instead of a formula expression. In that case, the elemental formula of that ensemble is used in the query, as computed by property `E_FORMULA`.

reaction expressions

Reaction expressions are the construct used for reaction substructure searches, for example when looking for certain bond transformations in a database of reactions. Obviously, the scanned file needs to contain reaction information for this to succeed.

An important aspect for reaction searches are atom mapping numbers, which link atoms in the reagent ensemble to the product ensemble, and likewise in the transformation scheme which needs to be matched. The central property for this is `A_MAPPING`. If this property is present, it is used to restrict matches to those reactions which embody a certain transformation, and are not a simple pair of ensembles which match substructures of the left and right part of the query transformation somewhere in their connectivity. Nevertheless, it is still possible to query reaction without a mapping scheme. That is identical to a pair of substructure searches. Also, individual parts of a reaction (the *reagent* and *product* ensembles, but potentially also the *catalyst* or *solvent* entries) can be used as targets for single-ensemble sub/super/full-structure searches via structure query expressions (see above).

A reaction expression is a list of three to six elements. The first element is always *reaction*, the second element the operator, and the third element the reaction source. The following operators can be used:

- =
  Reaction identity, i.e. full-structure reaction search. This is internally re-written to an equivalent hash code search as a property node.
- !=
  Reaction inequality, i.e. a negated full-structure reaction search. This is internally re-written to an equivalent hash code search as a property node.
- >=
  Reaction substructure search.
- >
  Reaction substructure search, excluding identity.
- <=
  Reaction superstructure search.
- <
  Reaction superstructure search, excluding identity.
- ~> or ~>=
  Reaction Tanimoto similarity search with a reporting threshold.
- %> or %>=
- Reaction Tversky similarity search with a reporting threshold.

Similar to structure query expressions, the operator can be modified by adding flag words as additional list elements to the operator list element. The following flags are recognized:

- *absolutestereo*
  Perform absolute stereo matching. By default, stereochemistry is not used in the query, except if set up explicitly as atom- or bond-specific query attribute in properties `A_QUERY` and `B_QUERY`. An alternative syntax is to directly attach an update *S* character to the operator.
- *allowmissingstereo*
  If set, absent stereochemistry descriptors in file structures can be matched by explicit stereo centers in the query structure. However, stereo center mismatches still lead to a match failure.
- *anyfragment*
  Report a match for full-structure search if any molecule of the file structure is identical to the query structure. For substructure/superstructure queries, this flag has no effect, since their default operation mode already covers the effects of the flag.
- *anyoverlap*
  If the substructure contains multiple fragments, they may match overlapping parts of the structure ensembles. By default, matched substructure fragments cannot overlap. This flag cannot be combined with *atomoverlap*.
- *atomoverlap*
  If the substructure contains multiple fragments, they may match overlapping atoms, but not bonds. By default, matched substructure fragments cannot overlap. This flag cannot be combined with *anyoverlap*.
- *bidirectional*
  If the query reaction does not match, try to match it also in the reverse reaction direction.
- *charge*
  Match formal charges of query atoms. By default, charges are not compared, except if set up explicitly as atom-specific query attribute in property `A_QUERY`.
- *emptyssismismatch*
  By default, a substructure without any atoms matches anything. If this flag is set, it matches nothing instead.
- *exactaro*
  Match aromatic bonds exactly. By default, simple single or double query structure bonds match structure file record aromatic bonds.
- *exactringsystem*
  Rings in substructure fragments must match complete ring systems only. For example, with this flag a benzene substructure no longer matches naphthalene, anthracene, etc. Non-ring parts of the substructure can still, if other query attributes do not prevent this, match both ring and chain parts of file structures. For full-structure queries, this flag has no effect.
- *extended*
  Use extended versions of the match procedures. For similarity queries, this enables the PubChem extended scoring mechanism. If the query structure is identical to a file structure both in stereochemistry and isotope labels, an artificial score of 104 is computed, 103 if

isotopes or stereochemistry match, but only one of these, 102 for basic equivalence of connectivity without isotopes or stereochemistry, and 101 for a tautomer. Compounds which are not structurally identical to the query structures using one of these criteria are scored normally.

- *framework*
  Substructure carbon atoms cannot have any unmatched, directly bonded carbon or hetero atom neighbors in the structure. Unmatched bonded hydrogen is allowed. This flag has an effect only for sub- and superstructure match modes.
- *implicitsinglearo*
  If this flag is set, bonds which were created with an implicit bond order when the query structure was decoded are matched as if they were explicit *single/aro* query bonds. This is a useful mode for emulating Daylight software.
- *isotope*
  Perform isotope matching. By default, isotope labels are not used in the queries, except if set up explicitly as atom-specific query attribute in property `A_QUERY`. An alternative syntax is to directly attach an *i* character to the operator.
- *matchallheavyatoms*
  Require that all heavy atoms in the file structures are matched. This feature generates matches of file structures similar to full-structure matches while allowing the use of substructures with variable match conditions, such as atom lists.
- *nobondorder*
  Do not compare bond orders. This flag has an effect only for sub- and superstructure match modes.
- *nochainonaro*
  Do not match chain parts of the query substructure on aromatic bonds in the file structures. This flag has an effect only for sub- and superstructure match modes.
- *nochainonring*
  Do not match chain parts of the query substructure on ring bonds in the file structures. This flag has an effect only for sub- and superstructure match modes.
- *nodoubleonaro*
  Do not match otherwise unmarked double bonds in the substructure onto aromatic bonds of the structures.
- *noquerytree*
  Deactivate extended matches requiring full checks of the query tree fields in the `A_QUERY` and `B_QUERY` properties in the query structures. Certain query inputs need these trees for precise matching, because the query cannot be expressed as a flat set of query attributes. Examples for queries requiring tree matching for proper execution are complex SMARTS expressions beyond those using only simple explicit or implicit *and* in atomic or bond expressions, and Recursive SMARTS. Disabling the flag may lead to a small speed-up for simple substructure queries.

- *noreactionflags*
  Do not match reaction transform flags in the substructure. If reaction flags are checked, which is the default for reaction queries but not for structure queries, both query and file structures need to have property `B_REACTION_CENTER` set for this to work. The supported set of comparisons is compatible with **MDL's ISIS** database. For standard reaction queries which check for specific bond changes, this flag should *not* be set.

- *nosingleonaro*
  Do not match otherwise unmarked single bonds in the substructure onto aromatic bonds of the structures.

- *nosubstructureh*
  For substructure match, ignore any hydrogens present in the query structure. This is a convenient shortcut to allow the use of hydrogen-complete structures as simple substructures. A similar scheme is automatically invoked for superstructure search, where hydrogens in the file structures are ignored in matching.

- *relativestereo*
  Perform relative stereo matching. By default, stereochemistry is not used in the query, except if set up explicitly as atom- or bond-specific query attribute in properties `A_QUERY` and `B_QUERY`. An alternative syntax is to directly attach a lowercase *s* character to the operator.

- *sethighlight*
  In case structure ensembles are retrieved from the file (**`molfile scan`** modes *ens, enslist, reaction or reactionlist*), the bonds and atoms matched by a substructure are marked in the returned structure-side ensembles with the highlight flags in properties `B_FLAGS` and `A_FLAGS`. In case multiple matches occur, the highlight set is an union of all processed matching substructure mapping. This flag is also automatically set if the data retrieval set in the **`molfile scan`** command includes related pseudo properties, such as *matchatoms* or *matchbonds*.

- *setmatchproperty*
  In case structure ensembles are retrieved from the file (**`molfile scan`** modes *ens, enslist, reaction or reactionlist*), the bonds and atoms matched by a substructure are marked in the returned structure-side ensembles by attached properties `A_SSMATCH` and `B_SSMATCH`. These are set to the labels of the matching substructure atoms or bonds. Unmatched structure ensemble parts have match property values of zero. In contrast to the *sethighlight* flag, this option attaches a new match property instance for every successful and processed match. Returned ensembles may therefore possess series of properties like `A_SSMATCH`, `A_SSMATCH/2`... and so on.

- *unique*
  Hint for the query processor that the query reaction is expected to be matched only once in the file, if at all. This is useful for query optimization. If a hit has been found, additional records need not to be checked.

The third mandatory parameter is the query reaction source. It can be any of

- A reaction handle
  The handle is decoded directly.

- A dataset handle
  A dataset containing at least one reaction. All dataset objects are checked, and internally for every reaction a separate expression node is created. The nodes are then linked via an *or* or *orcontinue* (in case a scoring operator is used) branch node. Dataset objects which are not reactions are silently ignored. The hydrogen status of the dataset reactions is not changed. In case there is only a single reaction in the dataset, this command is indistinguishable from using the reaction handle directly. In case the dataset does not contain any reactions, an error is raised.
- reaction line notation string
  A string representation of a reaction, in any format that can be decoded by the **`reaction create`** statement, for example a Reaction **SMILES**, **SMIRKS**, **RInChI** or a **Cactvs** serialized reaction object string. This query reaction is only temporarily instantiated and automatically deleted when the command finishes.

Reading one or more query reactions from a file handle directly in the query statement, as it is possible for structure queries, is currently not supported. Also, the tautomer match mode is not available for reaction matching because it interferes with atom map processing.

The optional query list items four to six are identical to those for structure query expressions. They represent a reporting threshold value and the *c1* and *c2* comparison algorithm parameters. Please refer to the paragraph on structure match expressions for more details.

The general approach to reaction sub- and superstructure matching is as follows:

- Perform bit vector screening for acceleration, if supported by the file format. The default reaction screen property is `X_SCREEN`. The name of the reaction screen bit property can be changed by setting the appropriate molfile handle attribute, so it is easily possible to use a custom reaction screen.
- Match the reagent side from the file record onto the reagent side of the query reaction, just like a structure query expression. If possible, structure screening (see paragraph on structure match expressions) is used as an acceleration filter in addition to the reaction screen.
- If atom mapping information is available, use it to set up a match constraint table for the product side, i.e. allow the product side substructure atoms with an atom mapping label which has a counterpart in a reagent substructure atom mapping value to match only the atom in the file product structure which has the same mapping label as the reagent side atom which was matched by the reagent substructure. For this to work, there need to be two matching pairs of mapping values on the reaction substructure and file reaction, though they of course can be different in both reactions. In case a 1:1 relationship cannot be established for an atom, the matching of this atom is not restricted.
- Match the product side, using mapping constraints where possible, and also using structure screens if available.
- If any of the previous steps fail, abort the sequence early, but if *bidirectional* matching is allowed, try again with the roles of the reaction substructure reagent and product ensembles swapped.

Besides the ensemble-level query attribute properties `A_QUERY` and `B_QUERY`, reaction matches also make use of `B_REACTION_CENTER` (for constraints on the type of transformation a bond undergoes) and `E_REACTION_ROLE` (for the identification of reagent and product ensembles in the reaction object).

Reaction similarity queries use the reaction screen set (by default, property `X_SCREEN`) instead of the structure screen that is used for structure similarity. This operation returns a single score. There is no scoring of the reagent or product ensembles.

Full-structure reaction matches are performed via hash code checks both the reagent and product sides. Atom mapping information is not used for this query operation. The suitable hash code is automatically selected depending on the operator modifiers (stereo, isotopes).

Starting with version 3.358 of the toolkit, property query expressions where the data type of the property is *reaction* are automatically parsed as reaction expressions.

Scan modes

The return value of the **`molfile scan`** command depends on the query mode. The default mode is *enslist* for the **`molfile scan`** command, but may be different when scanning other objects, such as datasets, networks or tables. The following modes are supported for file queries via the **`molfile scan`** command. Scan modes for other objects may include specific additional modes, while disallowing others.

- *array* (or alias *tclarray, dict, pythondict*)
  The mode parameter is a list consisting of the mode selector *array* and a nested list of properties and pseudo-properties. Each property item can be a list of one to three elements. The first element is a property or pseudo-property, the second element a name, and the third element again a property or pseudo property. The the second property item list element is omitted, the name is the same as the first element. If the third element is missing, it is assumed to be the pseudo-property *record.*

  In this mode, the **`molfile scan`** command returns a list of the names of the created arrays. For each name, a global **TCL** array variable or **PYTHON** dictionary is created, and for each match, a **TCL** array element with an element name equal to the value of the first item specification index and an element value equal to the value of the third item specification is created (or a dictionary entry with key and value for **PYTHON**). For example, the scan mode specification

  ```
  {array {E_NAME name2rec} {record rec2name E_NAME}}
  ```

  results in the creation of two global **TCL** arrays or **PYTHON** dictionaries in the current interpreter, called *name2rec* and *rec2name*. The first has array elements (for **PYTHON**, dictionary keys) where the element name is the name of the matching structure (property `E_NAME`), and the value the file record number (because it is the default). The second array has elements where the record number is the array element name, and the corresponding value the structure name. The return value of the scan statement is the list (tuple for **PYTHON**) *“name2rec rec2name”*, containing the names of the two variables created.

If array or dictionary elements for a specific key already exist, the new value is appended as a list or tuple object. The result registration procedure does not overwrite the existing content. So, for example in above case, if there are multiple records with the same structure name, the array element indexed by name would contain a list or records, not just a single record. Since the global arrays or dictionaries are persistent, data is also appended over multiple scan statements. If this is not desired, a statement like **`unset -nocomplain $arrayname`** should be executed before the scan is started. It is legal to use the same array or dictionary name for the registration of multiple properties. In this case, each match appends a new list element for every reported property, though these lists will not be nested.

- *bitvector*
  Return a bit vector (series of 0s and 1s in compact format) indicating the match status of every visited record. Internally, these bit are stored efficiently in 32-bit words. To the scripting interface they appear as a sequence of 0 and 1 without a space separator.
- *boolean*
  Return a boolean value indicating whether the next record matches or not.
- *booleanvector*
  Return a boolean vector (series of 0s and 1s as vector elements) indicating the match status of every visited record. The difference to the *bitvector* mode is that in the scripting interface the vector elements are already isolated elements, for example they appear space-separated in the string form.
- *count*
  Just count the number of hits, but do not report details. The result value is an integer.
- *delete*
  Delete hits from the file, if this is possible. This operation is performed after the scan has completed, not during the scan, so that file record numbers etc. do not change within a query.
- *ens*
  Return the handle or reference of the first matching ensemble. The query is stopped at that point. If no hits are found, an empty string is returned.
- *enslist*
  Return the handles or references of all matching ensembles. If no hits are found, an empty list is the result.
- *exists*
  Return a boolean flag indicating whether any hit exists. This is very similar to the *count* mode, except that query processing is stopped after the first match.
- *index*
  The file position index of the first matching object. This is the same as the *record* mode, except that each hit value is one less, since indices start at zero. The query is stopped after the first hit.
- *indexlist*
  A variant of the *recordlist* mode. The returned values are one less than the records, since indices start at zero.

- *molfile*
  The mode parameter list consists of the mode selector *molfile* and a structure file handle or reference, which must have been opened for writing, appending, or updating. The first matching structure is written to the file.After this, the query stops. The output file attributes determine format, selection of data written, structure encoding conventions such as hydrogen status, etc. If no matching structure is found, nothing is written. In this mode, the return value of the command is the matching record number of the input file, just as in the *record* mode.

- *molfilelist*
  The mode parameter is a list consisting of the mode selector *molfilelist* and a structure file handle or reference, which must have been opened for writing, appending, or updating. Matching structures are written to that file. The output file attributes determine format, selection of data written, structure encoding conventions such as hydrogen status, etc. If no matching structures are found, nothing is written. This mode is also implicitly selected if a structure file handle is directly provided as mode argument. In this mode, the return value of the command is a list of the matching record numbers of the input file, just as in the *recordlist* mode

- *property*
  The mode parameter is a list consisting of the mode selector *property* and a sequence of names (or references, for **PYTHON**) of properties and pseudo-properties. The selected properties for the first match are returned as a list. If there are no hits, an empty string is returned. The query stops after the first match.

- *propertylist*
  The mode parameter is a list consisting of the mode selector *propertylist* and a sequence of names (or references, for **PYTHON**) of properties and pseudo-properties. The selected properties for all matches are returned as a nested list. If there are no hits, an empty string is returned. This mode is also selected if the mode argument is simply a list of property and pseudo property names without an identifiable mode keyword as first list element.

- *reaction*
  Return the handle or reference of the first matching reaction. The query is stopped at that points. If no hits are found, an empty string is returned.

- *reactionlist*
  Return the handles or references of all matching reactions. If no hits are found, an empty list is the result.

- *record*
  The record number of the first file record which matches. In case a single physical file is searched, this is the same as *vrecord*, but if the scanned file is a virtual file consisting of multiple physical component files, this is the record number in the matching physical file. The scan is stopped when the first match has been found. If there are no matches, an empty string is returned.

- *recordlist*
  The same as the *record* mode, except that more than one match is potentially reported. In case a virtual file is searched, it is possible that duplicate values are returned, because the same record number from different physical files may be a hit. For unique record numbers, use the *vrecordlist* variant.

- *table*
  The mode parameter is a list consisting of the mode selector *table* and a sequence of properties and pseudo-properties. This scan mode returns a table handle or reference. The table is automatically configured with properly typed columns corresponding to the requested properties. For each hit, one row is added. If there are no hits, a table handle or reference is still returned, but the table does not have any rows. This retrieval mode is only available if the toolkit has been compiled with table support.

  The individual properties may also each be specified as a list consisting of the property name, and an arbitrary string. In that case, the string is used as the column name. By default, the column names are the same as the name of the property they store. Example:

  ```
  {table {E_NAME name} {E_CAS casno} record}
  ```

  sets up a table with three columns called *name*, *casno* and *record*. The first two columns contain property data from the matching file records, the last one the record in the file which matched.

  Instead of the keyword *table*, an existing table handle or reference may also be used. In that case, any existing matching table columns are automatically re-used to store result data. Additionally specified properties are added as new columns to the right of the previously existing columns. New table rows generated by matches are appended to the bottom of the table.

  The row names of added table rows are set to *Record%u*, with the file record number as variable part.

- *tablecollection*
  This mode is mostly identical with the *table* mode, and takes the same column specification parameters.The important difference is that this scan mode always retrieves the full objects associated with the filled table rows (ensembles or reactions). They are preserved and their relationship with the table marked. This can be useful if at a later stage in handling the table additional data needs to be computed or retrieved from an object. On the other hand this mode can be memory-intensive if many objects are created. Referral to associated objects may happen indirectly, for example with image columns where the exact image property is unknown until output time when the storage format is selected.

  The scan command mode returns the table handle or reference as result. The associated row objects are stored in the general namespace, and are not be a member of any dataset. They are visible like any other object of their type, for example via **`ens list`** or **`reaction list`** commands. Commands **`table ens`** and **`table reaction`** are useful to get the object subset associated with this table. Note that these table-associated objects are not automatically deleted when the table is destroyed - only their association is severed. If they are no longer needed, they should be destroyed explicitly.

- *vrecord*
  If the scan is executed on a single file, this is the same as *record*. In case a virtual file which consists of multiple physical files is searched, this is the virtual file record number, i.e. the overall record number in the concatenated component files.
- *vrecordlist*
  If the scan is executed on a single file, this is the same as *recordlist*. In case a virtual file which consists of multiple physical files is searched, this is a list of the virtual file record numbers, i.e. the overall record numbers in the concatenated component files.

If requested property data is not present on the object representing a hit, an attempt is made to compute it. If this fails, the retrieval modes *table* and *tablecollection* generate `NULL` cells, and property retrieval as list data produces empty list elements, but no errors. For minor object properties, the property list retrieval modes produces lists of all object property values instead of a single value. In *table*-based mode, only the data for the first minor object associated with the major object is retrieved, which makes this mode less suitable for direct minor object property retrieval.

Pseudo properties for retrieval

The following pseudo properties can be retrieved in *property/properylist* scan modes or as *table* values, in addition to standard property data:

- *avgscore*
  The average value of all computed scores, such as Tanimoto or Tversky similarity scores, in the matching query for this result.
- *conformerindex*
  The index of the matching conformer in case of 3D queries with multiple conformations, -1 if no matching conformer index was determined.
- *conformer*
  A list of the atomic coordinates of the matching conformer, if a 3D query was performed. If this is not the case, an empty vector is the result. The data type of this vector is *coorvec* (x,y,z-triples as vector elements).
- *filename*
  The name of the physical file the match occurred in. For normal, single-file scans, this is not interesting. However, for virtual files, only the combination of the pseudo properties *filename* and *record* is a complete reference.
- *image*
  A structure **GIF** image (property `E_GIF`) with highlighted matching substructure atoms and bonds. A normal `E_GIF` retrieval property would just show the structure, but without highlighting. The data type of this property is the same as that of `E_GIF` (depending on the configuration, a *diskfile* reference or an in-memory *blob*).
- *index*
  This is the same as *record*, except that the value is one less, since indices start with zero.

- *matchatoms*
  An integer vector holding the labels of all atoms matching the substructures used in evaluating the query expression. If no substructure was used for the match, this vector is empty. *highlighatoms* is an alias for this pseudo property.
- *matchbondatoms*
  The same as *matchbonds*, except that each element is a pair of the labels of the matching atoms in the bonds, not the bond label as a single number.
- *matchbonds*
  An integer vector holding the labels of all bonds matching the substructures used in evaluating the query expression. If no substructure was used for the match, this vector is empty. *highlightbonds* is an alias for this pseudo property.
- *matchcount*
  The first element of the *matchcounts* array, as described below. If the query does not contain any substructure match nodes, the result is empty.
- *matchcounts*
  An integer vector holding the number of distinct substructure matches for substructure query nodes in the query tree. For normal substructure expressions, this value can only be zero or one because the standard substructure match mode only checks for the presence of any match (match mode *first*). Additionally, this value can be minus one if the node was never evaluated, for example because it is part of an *or* expression. Only if the *count* modifier is used together with the substructure query operator, or the substructure operator is the range operator, the possibility of multiple matches is evaluated and larger values can be obtained. For these operations the match mode is currently always *distinctinneratoms* (see `match ss` command).
- *matchmask*
  A bitvector indicating which children of the root query node have matched. The length of the bitvector is the same as the number of children of the root node. A typical application of this retrieval item is in combination with a *range* node as root node.
- *maxscore*
  The maximum value of all computed scores, such as Tanimoto or Tversky similarity scores, in the matching query for this result.
- *merit*
  For queries which use a merit/demerit rating scheme (for example, Bruns/Watson queries) this retrieves the accumulated merit/demerit sum of the top-level query node. The query needs to match for this retrieval to work, so in case none of the demerit rules match, you get an empty result, not a default zero merit/demerit value. Internally, there is no distinction between merit and demerit scores. The keyword *demerit* is an alias for this pseudo-property.
- *minscore*
  The minimum value of all computed scores, such as Tanimoto or Tversky similarity scores, in the matching query for this result.

- *pass*
  The pass number of the query execution. Normal queries, i.e. those without *smartquery* nodes and without hand-crafted *passswitch* nodes are executed only once, and the pass is always zero.
- *parent*
  The parent structure of the matching structure as a packed, base64-encoded serialized object string. If the structure file does not contain a precomputed parent structure, or the main file structure contains it as property, it is computed from the main file structure as property `E_PARENT_STRUCTURE`.
- *productmatchatoms*
  The same as the *matchatoms* pseudo property, but for the ensemble on the right side of a matching reaction, not a simple structure. If no reaction was matched, this is an empty list.
- *productmatchbondatoms*
  The same as the *matchbondatoms* pseudo property, but for the ensemble on the right side of a matching reaction, not a simple structure. If no reaction was matched, this is an empty list.
- *productmatchbonds*
  The same as the *matchbonds* pseudo property, but for the ensemble on the right side of a matching reaction, not a simple structure. If no reaction was matched, this is an empty list.
- *reagentmatchatoms*
  The same as the *matchatoms* pseudo property, but for the ensemble on the left side of a matching reaction, not a simple structure. If no reaction was matched, this is an empty list.
- *reagentmatchbondatoms*
  The same as the *matchbondatoms* pseudo property, but for the ensemble on the left side of a matching reaction, not a simple structure. If no reaction was matched, this is an empty list.
- *reagentmatchbonds*
  The same as the *matchbonds* pseudo property, but for the ensemble on the left side of a matching reaction, not a simple structure. If no reaction was matched, this is an empty list.
- *record*
  The physical record number of the current physical file. For normal, single-file scans this is the same as the virtual record. For virtual files, this property needs to be combined with the *filename* pseudo property to obtain a complete reference.
- *rgatoms(rg)*
  A list of the atom labels in a matching structure which were mapped to an expanded R-group atom in the query. The property index is the name of the R-group of interest defined in the substructure, usually something like *R1*. If there was no expanded R-group of that name, the result list is empty.
- *rgattachments(rg)*
  A nested list of the atom label pairs of the bonds in a matching structure which connect between the structure framework and the atoms expanded as the named R-group *rg*. If there was no expanded R-group of that name, the result list is empty.

- *score*
  The first element of the *scores* array, as described below. If the query does not contain any scoring expressions, the result is empty.
- *scores*
  An integer vector of the results of all query expression branches, in depth-first left-to-right order, which computed a score, such as structure similarity queries with Tanimoto or Tversky bitvector comparisons, or the number of child branch hits of a *range* node. In case a branch was not executed when the match was determined, a zero value is reported.
- *structure*
  The dataset structure as a packed, base64-encoded serialized object string.
- *vrecord*
  The virtual record number. For single-file scans this is the same as the physical record number.

Record visitation order

The optional visitation order parameter, one of the optional query parameters listed in the next section, is primarily intended to be used for convenient execution of queries on a subset of records which were selected by a previous query on the same file. It can either be a numerical record list, with the first file record indicated as record one, or one of the keywords *sortup* or *sortdown*, followed by a property name. If this parameter is not set, or set to an empty string, or the magic string *all*, records are visited from the current input position in simple sequential order. If the query parameter dictionary additionally contains a *startposition* value, this start position refers to the index (plus one) of the first element of the specified record set, not to the original underlying file.

In the record list variant of this argument, the specified (virtual) records in the file are visited in the list order, and all other file records are ignored. For optimum performance, the records should be sorted in ascending order, but this is not necessary, and, since it does affect the order of the returned results, record visitation sets with record sequences in custom order sorted to some criterion can have uses. A suitable format for a record list is a saved result of **molfile scan** in the *recordlist* or *vrecordlist* scan modes. It is possible to use a sorted record list with a non-rewindable input file, but an unsorted list will fail in that case if the file input pointer needs to be positioned backwards.

The sort property option variant implies a visit of all file records, but in the order of the values of a property in that file, not the native record sequence in the file. Using this access method is not too much overhead for indexed file formats such as **CBS** or **BDB** with an index on the sort property, but a serious performance hit for standard text files. This method cannot be used with files which cannot be rewound and do not have the sort property data in some direct access field, since it requires a full pass through the file to gather the sort property data values before the actual query is processed.

Examples:

```
molfile scan $fh "structure >= C1NCCC1" vrecordlist \
   [dict create "order" [list 3 6 29 157]]
molfile scan $fh "structure ~>= $ehcmp 90" {table E_SMILES score} \
   [dict create "order" {sortup E_WEIGHT}]
```

Query parameters

The final optional parameter is a keyword/value list of various additional attributes for fine-tuning the execution of the query. The following keywords are recognized:

- *branchmaxhits number*
  Set a maximum number of hits for every branch below the root node of the query. This is a per-branch version of the more commonly used *maxhits* option, which sets a global hit limit. By default the limit recursively applies to all nodes below the root, but this option is often combined with the *branchnodetype* option. If the latter is set, the limit only applies to those nodes which are of the specified type. Any nodes which have accumulated the specified maximum number of hits during the execution of a `scan` command no longer match regardless of the contents of additional records they are tested against. The hit count is increased whenever the branch returns a positive result, even if an overall positive match is not found because conditions in other branches are not met. The option can also be applied to logical nodes, such as *and* or *or*. In case of *or* nodes and in circumstances with similar optimization opportunities, the use of this option does not force the execution of lower branches if the match result of the node can already be determined by a partial testing of its branches, so the count may be less than expected.
- *branchnodetype nodeclass*
  This option is useful only in combination with a *branchmaxhits* option. If it is set to the name of a query node class, such as *structure*, *reaction*, *formula* or *property*, the limit only applies to those nodes below the root node which are of the specified type.
- *cellobjectcolumn colname*
  This parameter can only be used if the output of the scan is to a table object. It is the name of a column which contains decodable structure or reaction data and should be parsed if a cell or row object is to be automatically generated for a scan result row.
- *fullblockscan auto/no/yes*
  Ths parameter can be set to the values *auto* (or -1), *no* (or 0) and *yes* (or 1). The default value is *auto*. If this flag is true, scanning does not immediately stop after the *maxhits* or *maxscan* limits have been reached. Instead, each query thread completes its currently allocated file section, but not pick up more work afterwards. This guarantees that a subsequent query on the file can resume after the last visited record, without omitting to test records in file sections where the threads did not complete their task. However, in the full block scan mode the *maxhits* and *maxscan* parameters are then only a guideline, since the threads will scan more records, and possibily generate more hits, until they have finished their block. In *auto* mode, the full block scan mode is active if more than one thread is actually spawned, and inactive when there is only a single thread processing the query.
- *keeptablecolumns 0/1*
  This parameter has only an effect if the output is written to a table object. By default, existing table columns are deleted from the result table before the retrieval fields are parsed. If the flag is set, they are retained.
- *keeptablerows 0/1*
  This parameter has only an effect if the output is written to a table object. By default, existing table rows are deleted from the result table before the scan commences. If the flag is set, they are retained.
- *matchcallback procname*
  The name of a Tcl procedure or Python function name or reference in the current interpreter which is called upon each match after processing all standard query conditions, as well as once each for initialization and finalization. In some scan modes, the function is also be

called to report a mismatch. The parameters passed to the function are the callback mode as a string (one of *init*, *match*, *mismatch* or *final*), the current number of hits, and a reference for the match results accumulation object. The format of the latter depends on the scan mode - for example, in scan mode *bitvector* it is the evolving string representation of the result vector, while in scan mode *propertylist* is is a nested list of property values extracted so far. The structure of the result accumulator is usually the same as the final result of the scan operation. Setting the procedure name to an empty string is the same as omitting this attribute - no TCL procedure is called.

- *maxhits number*
  The maximum number of hits to report. If this number has been reached, the scan stops. If it is set to a negative value, which is the default, an unlimited number of hits could be reported.
- *maxrecord number*
  The maximum record number to visit during the scan. If the file is shorter, this is no error. The scan can be resumed at the stop position.
- *maxscan number*
  The maximum number of records to scan in all query threads combined. If this number has been reached, the scan is stopped (but see *fullblockscan* parameter). If set to a negative value, which is the default, an infinite number of records could be scanned.
- *maxthreads number*
  The maximum number of threads to use for scanning. By default, only a single thread is used. The use of multiple threads can significantly accelerate the processing of large input files, but for input data sets with less than 5K text records, or 50K binary and indexed records the overhead is likely to outweigh the gains. If multiple threads are used, a rule of thumb is that maximum acceleration on a sufficiently large file, with plenty of memory and no competing processor load, is observed with two query threads per processor core. If set to a negative value, the internally used maximum number of threads is adjusted to the number of visible processor cores and number of threads already internally spawned for other purposes. Because of the need to have multiple concurrent read positions for multi-threaded searching, files which cannot be rewound are always the processed by a single thread. If the query file has less records than the maximum number of threads multiplied by the thread block size, the actual number of threads used can be smaller then specified..
- *order recordorder*
  Specify a scan order. By default, the data records are visited in increasing sequence from the current start position. The format of the value part of this dictionary entry has been described in the previous section. For more information, please refer to the paragraph on the record order list.
- *passlimit number*
  The number of accumulated hits which will prevent the execution of another query pass, typically with relaxed match conditions, in *smartquery* expressions and similar constructs. The default value is one, i.e. no additional passes will be executed if there is at least one match.

- *progresscallback procname*
  The name of a **Tcl** function or **Python** function name or reference which is called regulary during the file scan. That function could, for example, update a progress bar. The arguments to that function are, in this order, the operation code (*init*, *scan*, *final*), the handle of the scanned object (a *molfile* handle for the `molfile scan` command), the current number of record scans performed so far, the hit count and finally the size of the scanned object (file record count, dataset element count) as record or element count. If the object size is not known, minus one is passed. If a progress callback argument has been specified, it is passed as an additional and final parameter.

  The *init* and *final* function calls are made only once each, and before respectively after any *scan* calls for the execution of this statement. The short form *callback* is an alias for this keyword. Setting the option to an empty string disables all progress callback function calls.

- *progresscallbackarg string*
  An arbitrary value which is passed as an additional, last parameter to the progress callback function.

- *progresscallbackfrequency number*
  The frequency of callback function invocations, measured as the number of records scanned between calls. If set to a negative value, the default is used (currently one call per 100K records scanned). If set to zero, the callback function is not called during the scan, but still for initialization and finalization. The short form *callbackfrequency* is an alias for this keyword.

- *sscheckcallback procname*
  The name of a **Tcl** procedure or **Python** function name or reference which is called after all preliminary checks of a substructure or superstructure match operation have succeeded. Records which are skipped by screening mechanisms or where standard sub/superstructure query attributes already exclude a match do not trigger a function call. This function can be used to add additional criteria to the query which cannot be expressed by standard means.

  The arguments passed to this function are, in this order, the substructure object handle, the structure object handle, a nested list with label pairs of all matched substructure and structure atoms, and a nested list with label pairs of all matched substructure and structure bonds. In case of superstructure searches, the roles of substructure and structure are reversed, i.e. the substructure handle and the listed atoms and bonds refer to the current structure read from the scanned data source. The check function should either return 1 for a successful final check, or 0, which leads to a rejection of the match. It is also possible to raise an error, which terminates the query with an error, or exit with a break, which terminates the query without an error.

  While the callback routine is free to perform any additional match analysis, it must neither delete the structure or substructure, nor change its connectivity (remove or add atoms and bonds), nor discard or invalidate any property data used in the matching process. The computation or setting of any additional property data on the substructure or structure ensembles is allowed.

- *startposition number*
  A specific record to begin the scan at. By default the scan begins at the current read position of the file, except when it is at **EOF**. In that case, the file is automatically rewound. If a record visitation order list is used, the start position parameter indicates the record list index plus one to use as first file record to visit, not the file record proper.
- *target datasethandle/remotedataset/#new*
  The value of this argument is a local or remote dataset handle or reference. If the result of the scan are ensembles (query modes *ens* or *enslist*), reactions (query modes *reaction* or *reactionlist*) or a table object (mode *table* or *tablecollection*), the object is moved to the specified dataset. In case the dataset is local, the move happens during the query, so that a different script thread could already begin further processing. Data transfer to remote datasets is performed in a single batch just before the query command finishes. For query modes which do not generate chemical objects, such as the *recordlist*, *property* or *count* modes, this parameter is ignored. The special target name **#new** creates a new dataset. In that case, the command output is the handle or reference of the new dataset, overriding other output modes.
- *threadblocksize number*
  If multiple threads are used, each thread processes a section of the file. If it completes the section, it will then request the allocation of a new section after the last section already allocated to any worker thread. If this parameter is set to a negative value, which is the default, a suitable thread block size is automatically determined from the file characteristics. It will then be typically a value between 10K and 100K records.

More typical examples

Examples:

```
molfile scan $fh {structure = c1ccccc1} recordlist
molfile scan $fh {E_WEIGHT < 100} {propertylist E_SMILES E_NAME E_WEIGHT}
molfile scan $fh {notnull E_CAS} {table E_SMILES E_CAS}
molfile scan $fh {structure ~>= c1nnccc1 90} {score record}
molfile scan $fh "and {structure >= $ehss} {formula >= N3}}" ens
```

Distributed queries

Molfile object handles can be configured to listen on specific ports for remote scan requests. The syntax of a remote scan request is the same as for a normal file. The only exception is the handle argument. The command is executed asynchronously. Since because of this no direct results are returned, the remote scans are typically of a type which yields network-transferable objects (modes *ens*, *enslist*, *reaction*, *reactionlist*, *table*) and specify a target dataset object on the local system.

On the local system, a typical set-up looks like this:

```
set dh [dataset create]
dataset set $dh port 10001
molfile scan $remotehost:10002 {structure >= c1ncccc1} \
   {table record E_NAME E_CAS} {} {target $localhost:10001 startposition 1}
while {![dataset tables $dh {} count]} {
   sleep 1
}
```

In above code, we first create a recipient dataset object, and configure it to listen on port 10001 for incoming **CACTVS** objects - we are expecting a table object as result later. We then issue the query for execution on the remote host, and wait until the table object containing the results has arrived.

On the remote server, the set-up could look like this:

```
molfile open $dbfile r port 10002
vwait
```

Here the database file is opened, and a port for incoming requests opened. The **`vwait`** **TCL** statement does nothing, but keeps the interpreter running, while waiting for and processing events such as incoming scan commands. In this sample set-up, the remote server needs to be started first, because otherwise the connection to the remote file fails on the client.

Since execution of remote queries is asynchronous, the client could issue multiple query requests to different remote handles and then wait until results from all these requests have been collected, or a timeout or other error condition has been reached. The results could arrive in any order. The scan commands for a group of servers could, for example, specify different start positions and maximum scan values for distributed searching of a big file, or could gather results from different small files. Additionally, the use of multiple scan threads could be requested on the server by passing appropriate parameters in the control section of the command. Nevertheless, only a singled remote scan command per **TCL** script thread is executed on the server at any time. If multiple scans need to be executed in parallel on a single server, a collection of script threads need to be created via the *Thread* package, and then every thread told to open its own port listener.

The mechanism for the reception of messages for remote scans on *molfile* handles which listen on ports is subtly different from the processing of commands sent to listening dataset objects. The execution of scans requires active collaboration of a **TCL** interpreter. Commands are only read and processed when the interpreter is idle, for example while sitting in a **`vwait`** or **`sleep`** statement. In contrast, dataset object listeners do not rely on **TCL** interpreters, and are implemented as independent threads. Remote dataset commands, such as **`ens move`** or **`dataset pop`** with a remote dataset handle, are therefore executed at any time when a mutex lock on the database object and other accessed objects can be secured.

molfile set

```
molfile set filehandle ?property/attribute value?...
molfile set filehandle attribute_dictionary
f.set(property,value,...)
f.set({property:value,...})
f.property = value
f[property] = value
```

A standard data manipulation command. It is explained in more detail in the section on setting property data. The alternative short form with the single dictionary argument is functionally equivalent to using the expanded dictionary as separate property and value arguments.

Examples:

```
molfile set $fhandle F_GAUSSIAN_JOB_PARAMS(link0) [list \
   "%chk=144__303_2EVE_PDB_Opt8.chk" "%mem=128MB" "%nprocshared=2"]
```

The command can also be used to set a broad range of object attributes. The list of attributes is documented in the section on the **`molfile get`** command.

If an attribute is set for a multi-file virtual file, in most cases the attributes are set for all the files in the set. Some attributes, such as the record, apply to the virtual handle only and their modification indirectly addresses only one physical file. An important example for this is the *record* attribute, which positions the record I/O pointer into the physical file which contains the record in the virtually concatenated file.

Example:

```
molfile set $fhandle record 2
```

Above command repositions the file read/write pointer to the second record.

This command supports a special attribute value syntax for manipulating bitset-type attributes (only attributes, not property values). If the first character of the argument is a minus character (-), the named bits in the set identified by the remainder of the argument are unset. If it is a plus (+), they are additionally set. With an equal sign (=), or no special lead character, the flag set replaces the old value. A leading caret character (^) toggles the selected bits.

Example:

```
molfile set $fhandle readflags +pedantic
```

molfile setparam

```
molfile setparam filehandle property ?key value?...
molfile setparam filehandle property dictionary
f.setparam(property,?key,value?...)
f.setparam(property,dict)
```

Set or update a property computation parameter in the metadata parameter list of a valid property. This command is described in the section about retrieving property data. The current settings of the computation parameters in the property definition are not changed.

The return value is the updated property computation parameter dictionary.

molfile show

```
molfile show filehandle propertylist ?filterset? ?parameterdict?
f.show(property=,?filters=?,?parameters=?)
Molfile.Show(filename,property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **`molfile get`** command. The difference between **`molfile get`** and **`molfile show`** is that the latter does not attempt computation of property data, but raises an error if the data is not present and valid. For data already present, **`molfile get`** and **`molfile show`** are equivalent.

The **PYTHON** class method is a one-shot command. The transient *molfile* created from the initialization items is automatically closed when the command finishes.

molfile skip

```
molfile skip filehandle ?recordcount?
f.skip(?records=?)
```

Skip records in a file opened for input. If the file pointer is at the beginning of a new record, this next record is the first skipped. If the file pointer is stuck in the middle of a record, for example because a **`molfile read`** command failed due to a file syntax error, the first record counted is the remainder of the current record. An attempt is made to re-synchronize to the beginning of the next record.

By default a single record is skipped. If the record count parameter is specified, more than one record can be skipped. Because of the partially read record re-synchronization feature, negative record counts are not allowed in this command. The **`molfile backspace`** and **`molfile set record`** commands can be used to go back in a file.

The command returns the number of the next record to be read. In case an attempt was made to position behind the end of a file, or a record re-synchronization failed, an error is reported.

molfile sort

```
molfile sort fhandle {{property ?direction ?cmpflags ?cmpvalue???}...}
   ?outfile/handle?
f.sort(sortby=,?output=?)
```

Sort the records in the file according to the values of one or more properties or property fields contained in the file records, or computable on the objects read from the file. The output are byte-for-byte identical images of the input records, not records reconstructed from read data objects.

The property sort set consists of o sequence of zero or more sort specification elements. Every specification element is parsed as a sublist, but only the first element therein is mandatory. This element is either a property name, a property field name, or one of the magic names *#record* or *record* (for the file record) or *#random* or *random* or *rnd* (for a random number assigned to that record). The optional sort direction element may be *up*/*ascending* or *down*/*descending*. The default sort direction is upwards. The third optional comparison flags parameter can be set to a combination of any of the values allowed with the **`prop compare`** command. The default is an empty flag set.

If a comparison value is supplied as fourth argument, the sort utilizes the comparison results of read file object property values against this value for ranking, not the direct comparison result between the read file object property values. This is for example useful when sorting according to a bitvector similarity value to an external structure.

The first property or magic name in the sort list has the highest priority. In addition to the specified properties, the original record number is implicitly added as tie breaker to yield a stable sort. This automatic value is always sorted upwards. If an empty property list is specified, the result is thus a simple file copy without record rearrangement. In order to randomize the record order in a file, use a single *#random* sort property.

The sort properties do not need to be already present in the file. If necessary, an attempt is made to compute these on the objects read from the file in the first pass. It is possible to sort on properties which are not of the object class read from the file, for example atom properties when ensembles are read, or ensemble properties when reactions are read. In that case, the record is output at the position determined by the lowest sort rank of the property of that object, for example the minimum or maximum value of all values of an atom property in an ensemble. Additional data instances of the property associated with a given record are ignored, so no record duplicates are output.

The optional output parameter can either be the handle or reference of an opened **Tcl** or **Python** channel, including standard output and standard error or the name of a (preferably new) file, or a pipe construct. Output is appended to this output channel. If the parameter is omitted, the output is first written to a temporary file, the original file deleted and the temporary file renamed to the original file. In that case, the original file handle is automatically re-opened for reading on the new file. The input file handle must be positionable, because file records are accessed twice, once for reading the sort data and once for copying the records out. Sorting from standard input, pipes or other non-rewindable sources is therefore not supported, and neither is the sorting of files which are not simple record sequences. Sorting such files is currently only possible by using explicitly scripted record data buffering mechanisms.

On Windows, output to an open **Tcl** file handle or **Python** file reference s not supported, except for the standard output and error channels.

The return value of the command is the number of records written. The position of the sort file handle is set to the same location as before the command.

Examples:

```
molfile sort $fh {{E_NAME up {dictionary nocase}}} dict.sdf
molfile sort myfile.sdf {{record down}}
set fhtcl [open "randomized.sdf" w]; molfile sort $fh {{random}} $fhtcl
molfile sort $fh {{A_ELEMENT down} {E_WEIGHT up}} "|gzip >heavy.sdf.gz"
```

The first example creates a new file *dict.sdf* which contains the remaining records in the file associated with the file handle sorted by the value of property `E_NAME` in case-insensitive dictionary order. The second example reverses the order of the records in the file, replacing the original file in the process. The third example randomizes the record sequence in the original file, outputting the records in a new file which was opened for writing as a normal **Tcl** text file. The final example outputs a compressed SD file, with structures sorted by the heaviest element in the ensembles, and using the molecular weight as tie breaker.

molfile sqldget

```
molfile sqldget filehandle propertylist ?filterset? ?parameterdict?
f.sqldget(property=,?filters=?,?parameters=?)
Molfile.Sqldget(filename,property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **`molfile get`** command. The differences between **`molfile get`** and **`molfile sqldget`** are that the latter does not attempt computation of property data, but initializes the property value to the default and returns that default, if the data is not present and valid; and that the **SQL** command variant formats the data as **SQL** values rather than for **Tcl** or **Python** script processing.

The **Python** class method is a one-shot command. The transient *molfile* created from the initialization items is automatically closed when the command finishes.

molfile sqlget

```
molfile sqlget filehandle propertylist ?filterset? ?parameterdict?
f.sqlget(property=,?filters=?,?parameters=?)
Molfile.Sqlget(filename,property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **`molfile get`** command. The difference between **`molfile get`** and **`molfile sqlget`** is that the **SQL** command variant formats the data as **SQL** values rather than for **Tcl** or **Python** script processing.

The **Python** class method is a one-shot command. The transient *molfile* created from the initialization items is automatically closed when the command finishes.

molfile sqlnew

```
molfile sqlnew filehandle propertylist ?filterset? ?parameterdict?
f.sqlnew(property=,?filters=?,?parameters=?)
Molfile.Sqlnew(filename,property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **`molfile get`** command. The differences between **`molfile get and molfile sqlnew`** are that the latter forces re-computation of the property data, and that the **SQL** command variant formats the data as **SQL** values rather than for **Tcl** or **Python** script processing.

The **Python** class method is a one-shot command. The transient *molfile* created from the initialization items is automatically closed when the command finishes.

molfile sqlshow

```
molfile sqlshow filehandle propertylist ?filterset? ?parameterdict?
f.sqlshow(property=,?filters=?,?parameters=?)
Molfile.Sqlshow(filename,property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **`molfile get`** command. The differences between **`molfile get`** and **`molfile sqlshow`** are that the latter does not attempt computation of property data, but raises an error if the data is not present and valid, and that the **SQL** command variant formats the data as **SQL** values rather than for **Tcl** or **Python** script processing.

The **Python** class method is a one-shot command. The transient *molfile* created from the initialization items is automatically closed when the command finishes.

molfile string

```
molfile string enshandle/reactionhandle/datasethandle ?attribute value?...
molfile string enshandle/reactionhadle/datasethandle? ?attribute_dict?
Molfile.String(eref/xref/dref,?attribute,value?,...)
Molfile.String(eref/xref/dref,attribute_dict)
```

This command creates a byte vector representation of a structure file. The third argument in the **Tcl** variant (first for **Python**) is an ensemble, reaction or dataset handle or reference, not a file handle or reference as for other *molfile* commands.

If the selected output format module supports direct output into a string, the record image is created without intermediary forms. Otherwise, a anonymous temporary file is opened, the ensemble or reaction(s) written to that file, and the file content returned as string with all newlines etc.. The file is then removed.

Writing to binary formats is possible. The return value of the command is a byte vector, not a simple text string, so it may contain **NUL** bytes. By default, in the absence of an explicit format specification, a **MDL** *Molfile* is written.

The remaining parameters are interpreted as in the **`molfile set`** command. There are two equivalent command variants, either using attribute and value argument pairs or a dictionary as a single argument. The parameters in the extra arguments or dictionary are typically used to set a hydrogen status, select the output format, etc.

`molfile blob` is an alias to this command.

Example:

```
set jmestring [string trim [molfile string [ens create C1CC1] format jme]]
```

The example creates an input string for the popular JME Java structure editor by P. Ertl/Novartis. The **`string trim`** statement deletes the trailing newline. The necessary **JME** output module is automatically loaded if it is not already loaded or compiled-in when the format parameter is decoded.

String record representations generated by this command can be opened for input as string data with the *s* mode of the **`molfile open`** command:

```
set fh [molfile open [molfile string $eh] s]
```

molfile subcommands

```
molfile subcommands
dir(Molfile)
```

Lists all subcommands of the **`molfile`** command. Note that this command does not require a handle.

molfile sync

```
molfile sync filehandle
f.sync()
```

This command synchronizes the file contents with the file system. The I/O modules for most file formats automatically performs a simple file buffer flushing upon finishing the output of a record, so this command is needed only under special circumstances where complete file system synchronization is required, the file was written without immediate commits, the I/O module for the file format provides a special synchronization function, or the output was done via asynchronous I/O. In any case, every file is fully synchronized when it is closed, so calling this function for normal output operations is not required.

The command returns the original file handle or reference.

molfile toggle

```
molfile toggle filehandle
f.toggle()
```

Switch a file from input to output, or vice versa. If the file was in write, append or update mode when the command is executed, the file is rewound and the read pointer is now pointing to the first record, or the original end point for *append* files. If the file was configured for input, the file output mode is changed to *append* if the file is a normal file. If the file is a *scratch* file, the file is truncated to an empty file and the write position set to the first record.

Not all file types can be toggled. Special file types except FTP streams cannot, and it is not possible to toggle a simple disk file which was originally opened in *read only* mode (see **`molfile open`** command).

The command returns the original file handle or reference.

molfile transfer

```
molfile transfer filehandle propertylist ?targetpropertylist?
f.transfer(properties=,?target=?,?targetproperties=?)
```

Copy property data from one *molfile* object to another *molfile* object or other major object, without going through an intermediate scripting language object representation, or dissociate property data from the *molfile* object. If a property in the argument property list is not already valid on the source file object, an attempt is made to compute it.

If a target object is specified, the return value is the handle or reference of the target object. The source and target object cannot be the same object.

If a target property list is given, the data from the source is stored as content of a different property on the target. For this, the data types of the properties must be compatible, and the object class of the target property that of the target object. No attempt is made to convert data of mismatched types. In case of multiple properties, the source property list and the target property list are stepped through in parallel. If there is no target property list, or it is shorter than the source list, unmatched entries are stored as original property values, and this implies that the object class of the source and target objects are the same.

If no target object is specified, or it is spelled as an empty string or **PYTHON** **`None`**, the visible effect of the command is the same as a simple **`molfile get`**, i.e. the result is the property data value or value list. The property data is then deleted from the source object. In case the data type of the deleted property was a major object (i.e. an ensemble, reaction, table, dataset or network), it is only unlinked from the source object, but not destroyed. This means that the object handles returned by the command can henceforth the used as independent objects. They can be deleted by a normal object deletion command, and are no longer managed by the source object.

molfile truncate

```
molfile truncate filehandle ?record?
f.truncate(?record=?)
Molfile.Truncate(filename=,?record=?)
```

Truncate a file. If no explicit record is given, the file is truncated after the current record. In case the current record count of the file is less than the specified record, the command raises an error.

Only files which are rewindable can be truncated. In addition, the program must have write permission to the file, although it is not required that the file handle is opened for writing. The I/O modules for files formats which are not a simple record sequence must provide a truncation function or the operation will fail.

The command returns the original file handle or reference.

The **PYTHON** class method is a one-shot command. The transient *molfile* created from the initialization items is automatically closed when the command finishes.

molfile unlock

```
molfile unlock filehandle propertylist/molfile/all
f.unlock(property=)
```

Unlock property data for the file object, meaning that they are again under the control of the standard data consistency manager.

The property data to unlock can be selected by providing a list of the following identifiers:

- Property names or references
  Valid property instances on the file object are unlocked. Non-existent data is silently ignored. It is not possible to unlock individual property fields.
- *all*
  All valid file object properties are unlocked.
- *molfile*
  This is an object class identifier. All property data which is controlled by the *molfile* major object and attached to the specified object class is unlocked. Since files do not incorporate minor objects, this identifier is equivalent to *all*.

Property data locks are obtained by the **`molfile lock`** command.

This command is a generic property data manipulation command which is implemented for all major objects in the same fashion and is not related to disk file locking. Disk file locks can be set or reset by modifying the *molfile* object attribute *lock*. This is explained in more detail in the paragraph on the **`molfile get`** command.

The return value is the original *molfile* handle or reference.

molfile upgrade

```
molfile upgrade filehandle
f.upgrade()
Molfile.Upgrade(filename)
```

If the I/O module provides a function to upgrade the format of an older file to the latest version of the format, for example after a support library upgrade, that function may be used. The only format which currently supports this feature is **BDB**.

The command returns the original *molfile* handle or reference.

The **PYTHON** class method is a one-shot command. The transient *molfile* created from the initialization items is automatically closed when the command finishes.

molfile valid

```
molfile valid filehandle propertylist
f.valid(property/propertysequence)
```

Returns a list of boolean values indicating whether values for the named properties are currently set for the structure file. No attempt at computation is made.

Example:

```
if [molfile valid $fhandle F_COMMENT] {...}
```

`molfile has` is an alias to this command.

molfile vappend

```
molfile vappend filehandle objectlist
f.vappend(objectref/objectrefsequence)
```

Virtually append records to an open file handle. The underlying file is not modified, but all future input operations on this file behave as if the extra records were present.

Because no actual output is generated, this command can only be applied on files opened for *reading*, not output files. In addition, the file handle needs to refer to a normal disk file and to support going backwards in the file, i.e. this command cannot be used on structure files opened via **URL**s, standard I/O channels, socket connections or composite virtual files with multiple physical files or the contents of a directory. The file format must support multiple records and the records must be encoded as a simple concatenated byte sequence. Examples for formats which work are **SMILES** or **SD** files for structures, or **RXN** or **RD** files for reactions.

The object list may contain ensemble, reaction or dataset handles. The data is split into virtual records according to the storage capabilities of the file. The format of the data written to the virtual records can be controlled by setting the *writelist*, *droplist* and *hydrogens* status attributes on the file handle.

When executed for the first time on a file handle for which the record count is yet unknown, the existing file records must be tallied and all current physical record positions be registered. For very large files, this can take some time. However, this is not equivalent to reading the complete file, so it does not consume much memory and the command can in principle work on arbitrarily large files.

Virtual records are held as string images in memory. A couple of thousand such records should not be a problem for typical workstations, but for systematic editing of large files where every record is touched an explicit scripted input/output loop is preferable.

The return value is the new record count of the file.

Changes to the file can be committed to disk by means of the **`molfile vrewrite`** command.

Example:

```
molfile vappend $fhandle [ens create c1ccccc1]
```

molfile vdelete

```
molfile vdelete filehandle recordlist
f.vdelete(record/recordsequence)
```

Virtually delete records from an open file handle. The underlying file is not modified, but all future input operations on this file behave as if the specified records had been deleted.

Because no actual output is generated, this command can only be applied on files opened for *reading*, not output files. In addition, the file handle needs to refer to a normal disk file and to support going backwards in the file, i.e. this command cannot be used on structure files opened via **URL**s, standard I/O channels, socket connections or composite virtual files with multiple physical files or the contents of a directory. The file format must support multiple records and the records must be encoded as a simple concatenated byte sequence. Examples for formats which work are **SMILES** or **SD** files for structures, or **RXN** or **RD** files for reactions.

When executed for the first time on a file handle for which the record count is yet unknown, the existing file records must be tallied and all current physical record positions be registered. For very large files, this can take some time. However, this is not equivalent to reading the complete file, so it does not consume much memory and the command can in principle work on arbitrarily large files.

The record list is a list of integer values, with one as the first file record. The list does not need to be sorted, and duplicate record numbers or record numbers out of range are ignored. It is possible to virtually delete file records which are themselves virtual, i.e. were added by the *vappend*, *vreplace* or *vinsert* subcommands and are not physically present in the file.

Virtually deleted records have negligible memory demands, but will slightly slow down input operations on edited files.

The return value is the new record count of the file.

Changes to the file can be committed to disk by means of the **`molfile vrewrite`** command.

Example:

```
molfile vdelete $fhandle [list 3 9 6]
```

molfile verify

```
molfile verify filehandle property
f.verify(property)
```

Verify the values of the specified property on the *molfile* object. The property data must be valid, and a *molfile* property. If the data can be found, it is checked against all constraints defined for the property, and, if such a function has been defined, is tested with the value verification function of the property.

If all tests are passed, the return value is boolean 1, 0 if the data could be found but fails the tests, and an error condition otherwise.

molfile vinsert

```
molfile vinsert filehandle objectlist
m.vinsert(objectref/objectrefsequence)
```

Insert virtual records for the specified objects into the file. The insertion position is before the current read position.

Except for the difference in the location where the virtual records are inserted, the command is equivalent to the **`molfile vappend`** command and has the same features and limitations. Please refer to that command for details.

The return value is the new record count of the file.

Changes to the file can be committed to disk by means of the **`molfile vrewrite`** command.

molfile vreplace

```
molfile vreplace filehandle objectlist
m.vreplace(objectref/objectrefsequence)
```

Insert virtual records for the specified objects into the file. The current input record is virtually overwritten.

Except for the difference in the location where the virtual records are inserted, and the fact that an existing record is replaced, the command is equivalent to the **`molfile vappend`** command and has the same features and limitations. Please refer to that command for details.

It is possible to replace a record which is itself virtual, i.e. was introduced by a *vappend*, *vinsert* or *vreplace* subcommand. If more than one output object is passed, or the object is written as multiple file records, additional virtual records are created and the record count of the file increased accordingly.

The return value is the new record count of the file.

Changes to the file can be committed to disk by means of the **`molfile vrewrite`** command.

Example:

```
set eh [molfile read $fh]
ens expand $eh
molfile backspace $fh
molfile vreplace $fh $eh
ens delete $eh
```

This command sequence virtually replaces a record with a version where superatoms are expanded.

molfile vrewrite

```
molfile vrewrite filehandle ?filename?
m.vrewrite(?filename=?)
```

Commit all virtual record additions, deletions or replacements to a physical file. If no file name is given, the current file name is used. After writing, the file handle remains valid. It is open for reading, and positioned before the first record. At this moment, the file no longer contains any virtual modifications, but the file handle may again be subjected to virtual edit operations. In case a file name is specified, and is not the same as the name of the current file, the file handle refers to the new file when the command has finished.

All valid records are copied verbatim to the new file, without going through decoding and re-encoding or records (see **`molfile copy`** command). A temporary file in the same directory as the current file is created, and sufficient disk space needs to be present to hold both the original file and the edited version at the same time. In case a problem occurs, the temporary file is deleted and the current file remains active. Only if all write operations succeed the old file is deleted and the temporary file renamed if necessary. In case a file name is specified, and it is not the same as that of the current file, the original file remains untouched, but is no longer linked to the *molfile* handle. For large files, this operation can take some time because massive amounts of data may need to be moved.

If the file referenced by the file handle has not been edited with virtual record operations (*vappend, vdelete, vinsert, vreplace*), the command does nothing and is equivalent to a **`molfile rewind`**.

The command returns the number of records written.

Example:

```
set fh [molfile open „myfile.sdf“]
molfile vinsert $fh 1 [ens create c1ncccc1]
molfile vrewrite $fh „myfile_with_pyrdine_inserted_in_rec_1.sdf“
```

molfile write

```
molfile write filehandle ?objecthandle/objecthandlelist?...
f.write(objectsequence/objectref,...)
Molfile.Write(filename,objectsequence/objectref,...)
```

This commands writes structure and reaction data to a file. Object handles may be ensemble handles, reaction handles, dataset handles, or *molfile* handles.

If an object is an input *molfile* handle, objects are read from the file until **EOF** is encountered if the output file supports multiple records. If the output file type is single-record, only the next record is read. The types of objects which are collected from the input *molfile* handle are dependent on its read scope. These objects are then treated as if they were used as parameter objects directly. Objects obtained via a *molfile* handle are automatically deleted after they have been written. If the input file is already at **EOF** when the command is executed, no objects are read, and no error is generated. However, this does not trigger the **NULL** record output handling described below, because the file object was specified as an argument.

The type of data which is actually written to the file depends on its format. A file opened for ensemble output can be fed with any type of handle. If reactions or datasets are passed, these are taken apart and written as individual records. If the output file is a reaction file, and an ensemble is passed, the reaction it is a member of is looked up and used as output object. If the ensemble is not a reaction ensemble, an attempt is made to store it as a plain ensemble outside any reaction. If the output routine rejects this, an error is raised. In case of datasets passed as objects for reaction output, the individual dataset objects (ensembles or reactions) are written, in combination with reaction reference substitution in case ensembles instead of reactions are found. For full-dataset output, it is legal to pass non-dataset objects. No dataset-level information is written and the objects stored as an anonymous dataset.

It is legal to supply no object handles at all. Normally, this means that simply no output is performed. However, I/O modules for specific file formats may support the output of special **NULL** records. In that case, the output function is called once without any objects. An example are **Gaussian** job files, which allow you to write records in multi-link files, where the computation instructions are taken from the file property `F_GAUSSIAN_JOB_PARAMS`, without supplying a structure record.

As part of the output process, new information may be computed on the objects. In case the active settings on the output *molfile* handle demand a structural change of an object, for example the addition or removal of hydrogen atoms, or the re-coding of ionic versus pentavalent nitro groups and similar functionality, the write objects are temporarily duplicated and these duplicates undergo the structure changes. The original output objects are never indirectly edited in their connectivity by this command.

The *writelist* attribute of *molfiles* may be set to a list of properties which should be included in the output. This has an effect only for file formats which support the storage of custom data values and which can cope with the data types of the listed properties. By default, no attempt is made to actively compute these properties for output. If they are not present in the input data, their output is silently omitted, or **NULL** values are written, depending on how the output format encodes these things. However, if the *computeprops* flag is set on the output *molfile*, an attempt for computation is made, and after output, the objects retain this additional data if the computation succeeds.

If the hydrogen set mode of the output *molfile* calls for a change in hydrogen status, the stage when these computations are performed depends on the hydrogen addition mode. If the output mode calls for potential hydrogen additions, the computations are executed after the addition - and this means, on the temporary duplicate, so the original object does not see the new property data. If the hydrogen mode does not change the hydrogen set, or potentially removes hydrogens, computations are performed on the original objects and then the object is potentially duplicated, with all its data, for hydrogen removal and output. In the latter case, the additional property data is visible on the original input objects.

The command returns a list of the object handles or references which were actually written to file. In cases like a reaction being split into ensembles, or a dataset taken apart, this is not necessarily the same object handle collection as the input object list. For output from an input *molfile* argument, the total number of objects written is returned instead, because the read objects are not retained.

The **PYTHON** class method is a one-shot command. The transient *molfile* created from the initialization items is automatically closed when the command finishes.

Examples:

```
molfile write "myfile.sdf" $eh1 $eh2
set fhandle [molfile open z.cbin w hydrogens add format cbin]
molfile write $fhandle $dset1
molfile write $fhandle $dset2
molfile close $fhandle
```

The first sample line uses the single-shot file operation feature of the **`molfile`** command. Instead of a *molfile* handle, a file name is passed, and that file is automatically opened, the output performed, and then the file is closed. Two ensembles are written with a single statement to the output file *myfile.sdf*. The desired file format is guessed from the file name suffix. No change in hydrogen status, etc. is performed, and no extra data is written out.

The next four example lines show how two complete datasets can be written to a native **CACTVS** toolkit binary file. Hydrogens are added to structures or reactions in the dataset - but the original dataset elements are not changed, since the addition is performed on temporary object duplicates. Also, the **CACTVS** binary format is requested explicitly by setting the *format* attribute. In this case, this is not really required, since the file format could also be guessed from the file name suffix. However, in case a non-standard file name suffix is used, formats must be specified explicitly, or the default format (**MDL SD**-file) is used. If the **CACTVS** binary file is later opened for reading with a read scope of *dataset*, all dataset elements plus the dataset-level property data can be recovered.

The *network* Command

The **network** command is the major object command for network objects. These are a high-level abstraction for any kind of data relationship which can be expressed as a network of vertices and connections. Network objects do not have a direct relationship with ensemble or reaction objects, but of course network nodes or connections can store ensemble or reaction data. However, these are then stored as vertex or connection properties of the respective type and not as direct memberships, like ensembles or reactions which are a member of a dataset object.

The syntax of this command follows the standard schema of **command/subcommand/majorhandle**. Networks are major objects and thus do not need any minor object labels for identification.

There are currently no transient network objects.

Examples:

```
network get $nhandle N_CONNECTION_COUNT
```

This is the list of currently officially supported subcommands:

network append

```
network append nhandle ?property value?...
n.append({?property:value,?...})
n.append(?property,value,?...)
```

Standard data manipulation command for appending property data. It is explained in more detail in the section about setting property data.

The command returns the first data value.

Example:

```
network append $nhandle N_NAME "_new"
```

network assign

```
network assign nhandle srcproperty dstproperty
n.assign(srcproperty=,dstproperty=)
```

Assign property data to another property on the same network. Both properties must be associated with the network object class. This process is more efficient than going through a pair of **network get/network set** commands, because in most cases no string or TCL/PYTHON script object representations of the property data need to be created.

Both source and destination properties may be addressed with field specifications. A data conversion path must exist between the data types of the involved properties. If any data conversion fails, the command fails. For example, it is possible to assign a string property to a numeric property - but only if all property values can be successfully converted to that numeric type. The reverse example case always succeeds, out-of-memory errors and similar global events excluded.

The original property data remains valid. The command variant **network rename** directly exchanges the property name without any data duplication or conversion, if that is possible. In any case, the original property data is no longer present after the execution of this command variant.

If the properties are not associated with networks (prefix N_), the operation is performed on all network nodes or vertices if appropriate.

The command returns the object handle for **Tcl**, or object reference for **Python**.

Examples

```
network assign $nh V_IDENT V_NAME
```

network connections

```
network connections nhandle ?filterset? ?filtermode?
n.connections(?filters=?,?mode=?)
```

Standard cross-referencing command to obtain the labels or references of the connections the network contains as minor objects. This is explained in more detail in the section about object cross-references.

Examples:

```
network connections $nhandle
filter create isa_link property C_ONTOLOGY_LINK value is_a operator =
network connections $nhandle isa_link
```

The example simply returns a list of the labels of the connections the network contains as minor objects. The second example restricts these to the subset where property `C_ONTOLOGY_LINK` has a specific value.

network create

```
network create ?attribute value?...
network create ?dictionary?
Network(?attribute,value?,...)
Network(attributedict)
Network.Create(?attribute,value?,...)
Network.Create(attributedict)
```

Create a new network object. All networks are created empty. The optional attribute keyword/value list or single-parameter dictionary is processed just as with a **`network set`** command.

The return value of the command is the new network object handle or reference.

network dataset

```
network dataset nhandle ?filterlist?
n.dataset(?filters=?)
```

Return the dataset handle or reference of the dataset the network is member of. It the network is not member of a dataset, or does not pass all of the optional filters, an empty string (or **`None`** for **Python**) is returned.

Example:

```
network dataset $nhandle
```

network defined

```
network defined nhandle property
n.defined(property)
```

This command checks whether a property is defined for the network. This is explained in more detail in the section about property validity checking. Note that this is *not* a check for the presence of property data! The **`network valid`** command is used for this purpose.

network delete

```
network delete ?nhandle/nhandlelist/all?...
n.delete()
Network.Delete("all")
Network.Delete(?nref/nrefsequence/nhandle?,...)
```

Delete network objects and all their associated vertices and connections. The special parameter *all* may be used to delete all networks currently registered in the application. Alternatively, any number of network handles may be specified for specific object deletions.

The command returns the number of deleted networks.

Example:

```
network delete all
network delete $nhandle
```

network dget

```
network dget nhandle propertylist ?filterset? ?parameterdict?
n.dget(property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **`network get`** command. The difference between **`network get`** and **`network dget`** is that the latter does not attempt computation of property data, but rather initializes the property values to the default and return that default if the data is not yet available. For data already present, **`network get`** and **`network dget`** are equivalent.

network dup

```
network dup nhandle ?dataset? ?position?
n.dup(?target=?,?position=?)
```

Duplicate a network with all minor objects and all attached data on the network object proper and its minor objects.

The duplicate network is placed into the same dataset as the source, if it is a member of a dataset. Specifying an explicitly empty dataset argument (or **`None`** for **Python**) places the duplicate outside any dataset, regardless of the dataset membership of the source network.

If the duplicate is moved to a dataset, it is appended to the dataset end by default. This happens also if the position parameter is explicitly specified as *end* or an empty string. Otherwise, the network is inserted at the given position, starting with 0. If the requested position is larger than the current size of the dataset, the network is appended.

Example:

```
network dup $nhandle
```

The command returns a new network handle or reference.

network exists

```
network exists nhandle ?filterlist?
n.exists(?filters=?)
Network.Exists(nref,?filters=?)
```

Check whether a network handle is valid. The command returns boolean 0 or 1. Optionally, the network may be filtered by a standard filter list, and if it does not pass the filter, it is reported as not valid.

Example:

```
network exists $nhandle
```

network expr

```
network expr nhandle expression
n.expr(expression)
```

Compute a standard **SQL**-style property expression for the network. This is explained in detail in the chapter on property expressions.

network filter

```
network filter nhandle filterlist
n.filter(filters=)
```

Check whether the network passes a filter list. The return value is boolean 1 for success and 0 for failure.

network get

```
network get nhandle propertylist ?filterset? ?parameterdict?
network get nhandle attribute
n.get(property=,?filters=?,?parameters=?)
n.get(attribute)
n[property/attribute]
n.property/attribute
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

Examples:

```
network get $nhandle {N_VERTEX_COUNT N_NAME}
```

yields the vertex count and name of the network as a list. If the information is not available, an attempt is made to compute it. If the computation fails, an error results.

For the use of the optional property parameter list argument, refer to the documentation of the **ens get** command.

Variants of the **network get** command are **network new, network dget, network jget, network jnew, network jshow, network nget, network show, network sqldget, network sqlget, network sqlnew,** and **network sqlshow.**

In addition to property data, the network object possesses a few attributes, which can be retrieved with the **get** command (but not by its related sister subcommands like **dget, sqlget,** etc.). Some of them are also modifiable via **network set.** These attributes are:

- *address_city*
  The city part of the author contact address.
- *address_country*
  The country part of the author contact address, following the ISO3166 standard.

- *address_state*
  The state part of the author contact address. Empty if not applicable.
- *address_street*
  The street address part of the author contact address. Includes floor, house number, etc.
- *address_zip*
  The **ZIP** code or other applicable postal code of the author contact address.
- *affiliation*
  The institution the author works for.
- *affiliationduns*
  The **DUNS** registration ID of the affiliated institution. This is primarily useful for US government projects.
- *affiliationurl*
  The **URL** of the affiliated institution.
- *author*
  The author of the network, as free-form string data.
- *authorurl*
  A **URL** with information on the author, or an empty string if unset.
- *category*
  A category string to be used if the network is stored in a repository.
- *classuuid*
  The base class **UUID** of this network object, as related to its authorship attribute set.
- *coords*
  If the toolkit was compiled with factory support, these are the coordinates of the object on its workbench, encoded as integer pair. This attribute can be changed.
- *deletable*
  Flag indicating whether the object can be deleted with a standard **`network delete`** command. This attribute is read-only. Networks which are, for example, property data values or a part of a **`molfile loop`** command cannot be deleted by standard means.
- *date*
  The date the network structure was defined.
- *doi*
  A digital object identifier for the network object content, if defined.
- *eolchars*
  The default **`EOL`** character(s) to use when writing the table to text files. The default is the standard platform **`EOL`** character(s).
  The magic strings *windows*, *mac* (both checked for the first three characters only) as well as *unix* and *linux* are automatically translated to the standard platform line terminators and not copied verbatim. Alternative names for these standard system encodings are *crlf, cr* and *lf.* The special value *default* resets the attribute to the platform-dependent default.
- *email*
  A contact email of the author.

- *failures*
  A list of properties for which computation failed on this object. This is a read-only attribute. Depending on configuration settings, this information may be used to block pointless attempts at re-computation of incomputable data.
- *footer*
  If the toolkit was compiled with factory support, this is the footer of the object icon on a workbench. This attribute can be changed.
- *gflags*
  If the toolkit was compiled with factory support, this is the currently set object icon rendering flag collection.
- *header*
  If the toolkit was compiled with factory support, this is the header of the object icon on a workbench. This attribute can be changed.
- *hidden*
  Flag indicating whether the object is hidden. This is not the same as the *invisible* state. This attribute is intended to be used for rendering selections. This attribute can be changed.
- *invisible*
  Flag indicating whether the object is invisible. This is not the same as the *hidden* state. An invisible object is no longer accessible via its handle. This is usually the case for objects which are scheduled for deletion, but still have lingering pointer references. This attribute is read-only.
- *infourl*
  A **URL** with information on the network object content, or an empty string if unset.
- *javaobject*
  If the toolkit was compiled with **JNI** support, this attribute reports the memory address of the **JNI** wrapper class instance, if it exists.
- *keywords*
  A list of keywords associated with the network object.
- *license*
  The license class associated with this network object. Setting the license to a standard type updates the associated **URL** with a standard location.
- *licenseurl*
  A **URL** with details about the network object license.
- *literature*
  A free-form literature reference.
- *modcount*
  Object structure modification count. This attribute is read-only.
- *mutexcount*
  The number of recursive mutex locks held for this object. Only supported on Linux.
- *name*
  A free-form network name as string.
- *orcid*
  The **ORCID** code of the author (see www.orcid.org).

- *path*
  The repository path for displaying hierarchical repository trees. This attribute is independent of any file system paths.
- *phone*
  A contact phone number of the author.
- *progress*
  A user-defined progress value intended to track the state of lengthy operations on the table. It is an integer between zero and one hundred and is initially set to zero. When the argument is set, it accepts a floating point value, but the stored value is automatically rounded to the next integer and forced into the 0..100 range.
- *pyobject*
  If the toolkit was compiled with **Python** support, this attribute reports the memory address of the Python wrapper class instance, if it exists. This attribute is read-only.
- *pyrefcount*
  If the toolkit was compiled with **Python** support, this attribute reports the reference count of the Python wrapper class instance, if it exists. This attribute is read-only.
- *record*
  The current iterator record. The pseudo records follow the vertex list, the first vertex is record one.
- *refcount*
  If the **Tcl** interpreter is using native **Cactvs** objects instead of string-based major object handles and integer-based minor object labels to identify toolkit objects, this returns the number of **Tcl** object references active for this network. This attribute is read-only.
- *references*
  Cross references of the network. This is a nested list of class **UUID**s and reference type tags.
- *regid*
  For registered data networks, the registration ID. Zero if this is a private network object.
- *scoped*
  A boolean object visibility control flag. If set, and global control flag `::cactvs(object_scope)` is also set, the object is visible only in the**Tcl** interpreter which set the scope flag and thus claimed it. Object list commands executed in other interpreters omit this object, and attempts to decode its handle in other interpreters will fail. The most common use of this feature is the hiding of persistent chemistry objects in scripted property computation functions.
- *selected*
  Flag indicating whether the object is selected. This attribute can be changed.
- *tooltip*
  If the toolkit was compiled with factory support, this is the tooltip of the object icon on a workbench. This attribute can be changed.
- *uuid*
  An automatically generated **UUID** globally identifying the object. This attribute is read-only, different for every object, and not dependent on its contents.
- *version*
  A version number of the network. This is a string in a 1.2.3 (or shortened) style.

- *versionuuid*
  The version **UUID** associated with this network object as per its authorship attributes.
- *x*
  If the toolkit was compiled with factory support, this is the x coordinate of the object on its workbench. This attribute can be changed.
- *y*
  f the toolkit was compiled with factory support, this is the y coordinate of the object on its workbench.This attribute can be changed.

network getparam

```
network getparam nhandle property ?key? ?default?
n.getparam(property=,?key=?,?default=?)
```

Retrieve a named computation parameter from valid property data. If the key is not present in the parameter list, an empty string is returned (**None** for **PYTHON**). If the default argument is supplied, that value is returned in case the key is not found.

If the key parameter is omitted, a complete set of the parameters used for computation of the property value is returned in dictionary format.

This command does not attempt to compute property data. If the specified property is not present, an error results.

network hierarchy

```
network hierarchy nhandle ?filterlist? ?root?
n.hierarchy(?filters=?,?root=?)
```

Return the hierarchy handle or reference of the hierarchy the network is part of. If the network is not member of a hierarchy, or does not pass all of the optional filters, an empty string or **None** for **PYTHON** is returned. By default, the hierarchy object which directly contains the network is returned. If the *root* flag is set, the root hierarchy object is reported instead, which is the same only if the hierarchy has only a single level.

Example:

```
network hierarchy $nhandle
```

network index

```
network index nhandle
n.index()
```

Get the position of the network in the object list of its dataset. If the network is not member of a dataset, -1 is returned.

network jget

```
network jget nhandle propertylist ?filterset? ?parameterdict?
n.jget(property=,?filters=?,?parameters=?)
```

This is a variant of **`network get`** which returns the result data as a **JSON** formatted string instead of **TCL** or **PYTHON** interpreter objects.

network jnew

```
network jnew nhandle propertylist ?filterset? ?parameterdict?
n.jnew(property=,?filters=?,?parameters=?)
```

This is a variant of **`network new`** which returns the result data as a **JSON** formatted string instead of **TCL** or **PYTHON** interpreter objects.

network jshow

```
network jshow nhandle propertylist ?filterset? ?parameterdict?
n.jshow(property=,?filters=?,?parameters=?)
```

This is a variant of **`network show`** which returns the result data as a **JSON** formatted string instead of **TCL** or **PYTHON** interpreter objects.

network list

```
network list ?filterlist?
Network.List(?filters=?)
```

Without a filter list argument, the command returns a list of the handles or references of all existing network objects.

If a filter list is specified, only those networks which pass all filters are listed. Filters may refer to the minor objects of networks (i.e. vertices and connections). In that case, a filter succeeds if any vertex or connection passes the filter.

Examples:

```
network list
```

network lock

```
network lock nhandle propertylist/objclass/all ?compute?
n.lock(property=,?compute=?)
```

Lock property data of the network, meaning that it is no longer controlled by the standard data consistency manager. The data consistency manager deletes specific property data if anything is done to the network which would invalidate the information. Blocking the consistency manager can be useful when building networks from components in a script. Property data remains locked until is it explicitly unlocked.

The property data to lock can be selected by providing a list of the following identifiers:

- Property names
  Valid property instances on the network or network minor objects are locked. If the boolean *compute* flag is set, an attempt is made to compute the property if it is not yet present. Otherwise, a request to lock non-existent data is silently ignored. It is not possible to lock individual property fields.
- all
  All valid network and network minor object properties are locked. The compute flag is ignored.
- *vertex*, *connection*, *network*
  These is are object class identifiers. All property data which is controlled by the network major object and attached to the specified object class is locked.

A lock can be released by a **`network unlock`** command.

network max

```
network max nhandle propertylist ?filterset?
n.max(property=,?filters=?)
```

Get the maximum values of the properties named in the *propertylist* parameter. The return value of the command is a list of the maximum property values.

While it is possible to work with network properties, this is pointless since there is only a single instance of a network property per network. Usually, vertex or connection minor object properties are tested. The objects whose property values are used for the determination of the maximum values may optionally be filtered by a standard filter set. If no objects pass the filter, the result is an empty list.

Example:

```
network max $nhandle V_LEVEL
```

computes the maximum value of the `V_LEVEL` property over all vertices.

network metadata

```
network metadata nhandle property ?field ?value??
n.metadata(property=,?field=?,?value=?)
```

Obtain property metadata information, or set it. The handling of property metadata is explained in more detail in its own introductory section. The related commands **`network setparam`** and **`network getparam`** can be used for convenient manipulation of specific keys in the computation parameter field. Metadata can only be read from or set on valid property data.

Valid field names are *bounds*, *comment*, *info*, *flags*, *parameters* and *unit*.

network min

```
network min nhandle propertylist ?filterset?
n.min(property=,?filters=?)
```

Get the minimum values of the properties named in the *propertylist* parameter. The return value of the command is a list of the minimum property values.

While it is possible to work with network properties, this is pointless since there is only a single instance of a network property per network. Usually, vertex or connection minor object properties are retrieved. The objects whose property values are used for the determination of the minimum values may optionally be filtered by a standard filter set. If no objects pass the filter, the result is an empty list.

Example:

```
network min $nhandle V_LEVEL
```

computes the minimum value of property `V_LEVEL` over all vertices of the network.

network move

```
network move nhandle ?datasethandle|remotehandle? ?position?
n.move(?target=?,?position=?)
```

Make a network a member of a dataset, or remove it from a dataset. If the dataset handle parameter is omitted, or is an empty string (or **None** for **PYTHON**), the object is removed from its current dataset. If it was not a dataset member, this command variant does nothing. The dataset handle may be the name of a remote dataset for moving objects over a network connection.

If a target dataset handle or reference is specified, the object is added to the dataset, if allowed by the acceptance bits of the dataset, and removed from any dataset it was member of before the execution of the command. By default the object is added to the end of the dataset object list, but the final optional parameter allows the specification of a dataset object list index. The first position is index zero. If the parameter value *end* is used, or the index is bigger than the current number of dataset objects minus one, the object is appended as per the default. It is legal to use this command for moving objects within the same dataset.

Another special position value is *random*. This value moves to the object to a random position in the dataset. Using this mode with remote datasets is currently not supported.

By default, datasets do not accept networks as objects. This must be explicitly enabled by modifying the acceptance bits, as for example in

```
dataset append $dhandle accepts network
```

The dataset handle cannot be a transient dataset.

The return value of the command is the dataset of the network prior to the move operation. It is either a dataset handle/reference, or an empty string (**TCL**) or **None** (**PYTHON**) if it was not member of a dataset.

This command interacts with the insert control mechanism of size-constrained datasets. More information is provided in the description of the *sizecontrol* dataset parameter.

Examples:

```
network move $nhandle $dhandle 0
network move $nhandle
```

In the first sample line, the network is inserted as the first element in a dataset. The second line reverts this operation and removes the network from the dataset.

This command can be used with a remote dataset descriptor. In that case, the network is packed into a serialized object representation, transmitted over the network and restored as member of the remote dataset at the specified position. The local network is deleted if the transfer succeeds.

Example:

```
network move $nhandle blockbuster@server2:9998 end
```

This command moves the network to the dataset which was set up as listener on port 9998 and pass phrase *blockbuster* on host *server2*. The local network is deleted, and its copy is inserted at the end of the remote dataset.

network mutex

```
network mutex nhandle mode
n.mutex(mode)
```

Manipulate the object mutex.

During the execution of a script command, the mutex of the major object(s) associated with the command are automatically locked and unlocked, so that the operation of the command is thread-safe. This applies to builds that support multi-threading, either by allowing multiple parallel script interpreters in separate threads or by supporting helper threads for the acceleration of command execution or background information processing.

This command locks major objects for a period of time that exceeds a single command. A lock on the object can only be released from the same interpreter thread that set the lock. Any other threaded interpreters, or auxiliary threads, block until a mutex release command has been executed when accessing a locked command object. This command supports the following modes:

- *lock*
  Increase the recursive mutex lock count on the object. The command returns the current lock count after the command, excluding the transient single-command lock.
- *reset*
  Release all persistent locks on the object, if they exist.
- *test*
  Return the current persistent lock count on the object. This excludes the transient per-command lock.
- *unlock*
  Decrease the recursive lock count on the object. The command returns the current lock count after the command, excluding the transient single-command lock. Unlocking an object which has not been persistently locked results in an error.

There is no *trylock* command variant because the command already needs to be able to acquire a transient object mutex lock for its execution.

The command returns the current lock count.

network need

```
network need nhandle propertylist ?mode? ?parameterdict?
n.need(property=,?mode=?,?parameters=?)
```

Standard command for the computation of property data, without immediate retrieval of results. This command is explained in more detail in the section about retrieving property data.

The return value is the original network handle or reference.

Example:

```
network need $nhandle V_LEVEL recalc
```

network new

```
network new nhandle propertylist ?filterset? ?parameterdict?
n.new(property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **`network get`** command. The difference between **`network get`** and **`network new`** is that the latter forces the re-computation of the property data, regardless whether it is present and valid, or not.

network nget

```
network nget nhandle propertylist ?filterset? ?parameterdict?
n.nget(property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **`network get`** command. The difference between **`network get`** and **`network nget`** is that the latter always returns numeric data, even if symbolic names for the values are available.

network nnew

```
network nnew nhandle propertylist ?filterset? ?parameterdict?
n.nnew(property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading object data and attributes. It is explained in more detail in the section about retrieving property data.

For examples, see the **`network get`** command. The difference between **`network get`** and **`network nnew`** is that the latter always returns numeric data, even if symbolic names for the values are available, and that property data re-computation is enforced.

network pack

```
network pack nhandle ?maxsize? ?requestprops? ?suppressedprops? ?compressionlib?
n.pack(?maxsize=?,?requestprops=?,?suppressedprops=?,?compressionlib=?)
```

Pack the network object into a base64-encoded compressed serialized object string. This string does not contain any non-printable characters and is a full dump of the internal state of the object, omitting only property data that was declared to be so easily re-computed that a dump is not worthwhile. The network vertex and connection minor objects and their property data are part of the dump.

The maximum size of the object string (default -1, meaning unlimited) can be configured by the optional *maxsize* parameter. The size is specified in bytes. If the pack string would be longer than the maximum size, an error results.

The two optional parameter lists allow to request a specific property set to be part of the package, even if it normally would not be included, and to explicitly omit properties from the dump. No property computation is performed, and suppressed properties are not purged from the source ensemble.

The default compression library is *zlib*. Other useful variants include *lzo* and *gzip* (and there are other internal types), but these may not be available on all builds due to license issues, and you need to specify the compression library when a dataset is unpacked. It is generally recommended to stay with *zlib*.

The command returns the pack string.

In **Python**, networks support the standard *pickle*/*unpickle* protocol.

network properties

```
network properties nhandle ?pattern? ?noempty?
n.properties(?pattern=?,?noempty=?)
```

Get a list of valid properties of the network proper and its vertex and connection minor objects.

The list can be filtered by a string match pattern. If the *noempty* flag is set, only properties where at least one data element controlled by the network (i.e. a value for an vertex of the network, etc.) is not the property default value are output. By default, the filter pattern is an empty string, and the *noempty* flag is not set.

Example:

```
network properties $nhandle N_*
```

network purge

```
network purge nhandle propertylist/network/vertex/connection ?emptyonly?
n.purge(?properties=?,?emptyonly=?)
```

Delete property data from the network. The properties may be network properties (prefix `N_`), or properties of the network minor objects, i.e. vertices (prefix `V_`) and connection (prefix `C_`). If a property marked for deletion is not found on the associated objects, it is silently ignored.

The optional boolean flag *emptyonly* allows to restrict the deletion to those properties where all the values of a property associated with a network object (such as on all vertices in a network for vertex properties, or just the single network property value for network properties) are set to the default property value.

In addition to property names, the object class names **`network`**, **`vertex`** or **`connection`** may be used. These delete all property data of that class from the network. They do not delete the objects proper, e.g. all vertices are still present after a **`network purge $nh vertex`**, though without any data that was not locked.

The return value is the original network handle or reference.

Examples:

```
network purge $nhandle N_NAME
network purge $nhandle V_ONTOLOGY_TERM 1
network purge $nhandke connection
```

network read

```
network read filename ?maxdefinitions?
Network.Read(filename=,?maxdefinitions=?)
```

Read network data from a file and create a new network object.

The filename argument can either be a disk file name, a pipe to an external generator program which starts with a vertical bar and is followed by the pipe programs and their parameters, the magic file name *stdin*, or, on Unix-class systems, an existing **Tcl** or **Python** file or socket handle.

Currently, there are seven network file formats which can be read. These are automatically identified by peeking into their contents, not by their file name. The recognized formats are:

- the native **CACTVS** network object dump file (standard suffix *.nbin*),
- **OBO** ontology definition files in version 1.2 (standard suffix *.obo*).
- **GENEONTOLOGY** ontology definition files (standard suffix *.go*)
- **SIF** network files (standard suffix *.sif*)
- **GML** network files (standard suffix *.gml*)
- **XGMML** network files (standard suffix *.xgmml* or *.gr*)
- **KEGG** metabolism networks (standard suffix *.kgml* or *.kegg*)
- **KNIME** network data files (standard suffix *.knet)*. These can only be read from a disk file, not a pipe or other data channel because they are internally a zipped collection of data files.

All of these formats can store only a single top-level network, so there are no record positioning commands for network data input. Nested networks, for example in the **GML** or **XGMML** formats, are currently not directly readable. Some additional network file formats can currently be written, but not read (for example, **GRAPHVIZ** *.gv* and **KNIME** *BEEF*).

The optional definition count argument can be used to stop reading a file after the specified number of vertex definitions have been read. This is basically a debug feature which allows program testing with a smaller network. The option is not supported with native **CACTVS** network dump files.

The command returns the handle or reference of the new network object.

Example:

```
set nh [network read "chebi.obo"]
```

network ref

```
Network.Ref(identifier)
```

PYTHON only method to get a reaction reference from a handle or another identifier. For networks, other recognized identifiers are network references, integers encoding the numeric part of the handle string, the **UUID** of the network object, or its name.

network rename

```
network rename nhandle srcproperty dstproperty
n.rename(srcproperty=,dstproperty=)
```

This is a variant of the **`network assign`** command. Please refer the command description in that paragraph.

network rewind

```
network rewind nhandle
n.rewind()
```

Reset the network iterator position. This is equivalent to setting the *record* network attribute to one.

network scan

```
network scan nhandle expression/queryhandle ?mode? ?parameterdict?
n.scan(query=,?mode=?,?parameters=?)
```

Perform a query on the network. The syntax of the query expression is the same as that of the **`molfile scan`** command and explained in more detail in its section about query expressions. In many aspects, this command behaves like an in-memory data file version of the **`molfile scan`** or **`dataset scan`** commands. The main difference is that the primary objects that are scanned are the vertices of the network, and their property data (or indirectly, the data of the network major object, or the directly attached connection minor objects) is what is checked when determining a match or mismatch.

The optional parameter dictionary is the same as for **`molfile scan`**, but not all parameters are actually used. At this time, only the *matchcallback, maxhits, maxscan, order, progresscallback, progresscallbackfrequency, sscheckcallback, startposition* and *target* parameters have an effect. In case a progress callback function is used, the network handle is passed as argument in place of the *molfile* handle in **`molfile scan`**.

The vertices of the network are visited in the order of their index value (see **`vertex index`** command). This index value plus one is used also used as a replacement for the record number of a vertex in the dataset.

The return value depends on the mode. The default mode is *vertexlist*. The following modes are supported for dataset queries:

- *array* (or alias *tclarray, dict, pythondict*)
  The mode parameter is a list consisting of the mode selector *array* and a nested list of properties and pseudo-properties. Each property item can be a list of one to three elements. The first element is a property or pseudo-property, the second element a name, and the third element again a property or pseudo property. The the second property item list element is omitted, the name is the same as the first element. If the third element is missing, it is assumed to be the pseudo-property *record*.

  In this mode, the command returns a list of the names of the created arrays. For each name, a global **Tcl** array variable or **Python** dictionary is created, and for each match, a **Tcl** array element with an element name equal to the value of the first item specification index and an element value equal to the value of the third item specification is created (or a dictionary entry with key and value for **Python**). For example, the scan mode specification

  ```
  {array {V_LABEL id2rec} {record rec2id V_LABEL}}
  ```

  results in the creation of two global **Tcl** arrays or **Python** dictionaries in the current interpreter, called *id2rec* and *rec2id*. The first has array elements (for **Python**, dictionary keys) where the element name is the label of the matching vertex (property `V_LABEL`), and the value the pseudo-record number (the default since the third list parameter is omitted). The second array has elements where the record number is the array element name, and the corresponding value the vertex label. The return value of the scan statement is the list (tuple for **Python**) *"id2rec rec2id"*, containing the names of the two variables created.

  If array elements for a specific key already exist, the new value is appended as a list object. The registration procedure does not overwrite existing content. Since global arrays are persistent, data is appended over multiple scan statements. If this is not desired. a statement like **`unset -nocomplain $arrayname`** should be executed before the scan is started. It is legal to use the same array name to register multiple properties in the array. In that case, any match appends a new list element for every property. The list is however not nested.

- *bitvector*
  Return a string-encoded bit vector (series of 0s and 1s) indicating the match status for every visited vertex.
- *boolean*
  Return a boolean value indicating whether the next vertex matches or not.
- *booleanvector*
  Return a boolean vector (series of 0s and 1s as vector elements) indicating the match status of every visited vertex. The difference to the *bitvector* mode is that in the scripting interface the vector elements are already isolated elements, for example they appear space-separated in the string form.
- *count*
  Count the number of hits. The result value is an integer.
- *delete*
  Delete hit vertices from the network. This is the only scan command which actually changes the network.
- *exists*
  A boolean check for the existence of a hit. The same as count, except that the scan stops after the first match.
- *file*
  The mode parameter is a list consisting of the mode selector *file* and a *molfile* handle, which must have been opened for writing, appending, or updating. The first matching vertex is written to the file. After that. the scan stops. File attributes determine format, selection of data written, structure encoding conventions such as hydrogen status, etc. If no matching vertex is found, nothing is written. This mode only works with structure files that can accept a vertex or a network object as output target. There are no I/O modules in the standard toolkit package which currently have this capability.
- *filelist*
  The mode parameter is a list consisting of the mode selector *file* and a *molfile* handle, which must have been opened for writing, appending, or updating. Matching vertices are written to that file. File attributes determine format, selection of data written, structure encoding conventions such as hydrogen status, etc. If no matching vertex is found, nothing is written. This mode will only work with structure files that can accept a vertex or a network object as output target. There are no I/O modules in the standard toolkit package which currently have this capability
- *index*
  This is the same as *record*, except that the returned value is one less, since indices start with zero.
- *indexlist*
  This is the same as *recordlist*, except that the returned value is one less, since indices start with zero

- *property*
  The mode parameter is a list consisting of the mode selector *property* and a sequence of properties and pseudo-properties. The selected properties for the first match are returned as a list. After the first match, the scan stops. If there are no hits, an empty string is returned. Properties which can be used in network scans are either of the *vertex* class (prefix V, the most common case for network queries), class *network* (prefix N, of limited value since there is only a single network scanned, and the property values are thus the same for all matches), or of the *connection* class (prefix C). Connection properties are reported as lists which contain the data of all connections the matching vertex participates in.

- *propertylist*
  The mode parameter is a list consisting of the mode selector *propertylist* and a sequence of properties and pseudo-properties. The selected properties for all matches are returned as a nested list. If there are no hits, an empty string is returned. This mode is also selected if the mode argument is simply a list of property and pseudo property names without an identifiable mode keyword as first list element.

- *record*
  Return the sequence number of the first hit. Sequence numbers begin, for the sake of comparability with structure file scan record numbers, with one and are equivalent to the vertex index position plus one. The scan stops after the first match.

- *recordlist*
  Return sequence numbers of all hits, or an empty list. Sequence numbers begin, for the sake of comparability with structure file scan record numbers, with one and are equivalent to the vertex index position plus one.

- *table*
  The mode parameter is a list consisting of the mode selector *table* and a sequence of properties and pseudo-properties. This scan mode returns a table handle. The table is automatically configured with properly typed columns corresponding to the requested properties. For each hit, a row is added. If there are no hits, a table handle is still returned, but the table does not have any rows. This retrieval mode is only available if the toolkit has been compiled with table support. The individual properties may also be specified each as a list consisting of the property name, and an arbitrary string. In that case, the string is used as the column name. By default, the column names are the same as the name of the property they store. Example:

  ```
  {table V_LABEL N_CONNECTION_COUNT C_LABEL C_ONTOLOGY_LINK}
  ```

  sets up a table with four columns that store the matching vertex label, the network connection count (the same value for all rows, since all matches are all on the same network), and the labels and link types of the connections attached to the matched vertex. The latter two columns are automatically stored as vector types, since there are potentially multiple values associated with a single matching vertex.

  Instead of the keyword *table*, an existing table handle may also be used. In that case, any existing matching table columns are automatically re-used to store result data. Additionally specified properties are added as new columns to the right of the previously existing columns. New table rows generated by matches are appended to the bottom of the table.

- *tablecollection*
  Since all objects subject to scans with this command are already in memory, this mode is identical to the *table* scan mode for network scans. No duplicate table reference objects are created. The result table always refers the network objects directly.
- *vrecord*
  For network scans, this is the same as *record*.
- *vrecordlist*
  For network scans, this is the same as *recordlist*.
- *vertex*
  Report the label of the first matching vertex. This is functionally equivalent to the mode **`property V_LABEL`**, though internally optimized. The scan stops after the first match.
- *vertexlist*
  Report a list of the labels of the matching vertices. This is functionally equivalent to the mode **`propertylist V_LABEL`**, though internally optimized.

If requested property data is not present on vertices to be tested, an attempt is made to compute it. If this fails, in retrieval mode *table* **`NULL`** cells are generated, and property retrieval as list data produces empty list elements, but no errors.

The following pseudo properties can be retrieved in addition to normal properties:

- *avgscore*
  The average value of all computed scores, such as Tanimoto, Cosine or Tversky similarity scores, in the matching query for this result.
- *conformerindex*
  The index of the matching conformer in case of 3D queries with multiple conformations, -1 if no matching conformer index was determined.
- *conformer*
  A list of the atomic coordinates of the matching conformer, if a 3D query was performed. If this is not the case, an empty vector is the result. The data type of this vector is *coorvec* (x,y,z-triples as vector elements).
- *filename*
  This pseudo-property is only provided for compatibility with **`molfile scan`**. It is always an empty string.
- *image*
  A structure **GIF** image (property `E_GIF`) with highlighted matching substructure atoms and bonds. A normal `E_GIF` retrieval property would just show the structure, but without highlighting. The data type of this property is the same as that of `E_GIF` (depending on the configuration, a *diskfile* reference or an in-memory *blob*).
- *index*
  This is the same as *record*, except that the retrieved value is one less, since indices start with zero.

- *matchatoms*
  An integer vector holding the labels of all atoms matching the substructures used in evaluating the query expression. If no substructure was used for the match, this vector is empty. *highlighatoms* is an alias for this pseudo property.

- *matchbondatoms*
  The same as *matchbonds*, except that each element is a pair of the labels of the matching atoms in the bonds, not the bond label as a single number.

- *matchbonds*
  An integer vector holding the labels of all bonds matching the substructures used in evaluating the query expression. If no substructure was used for the match, this vector is empty. *highlightbonds* is an alias for this pseudo property.

- *matchcount*
  The first element of the *matchcounts* array, as described below. If the query does not contain any substructure match nodes, the result is empty.

- *matchcounts*
  An integer vector holding the number of distinct substructure matches for substructure query nodes in the query tree. For normal substructure expressions, this value can only be zero or one because the standard substructure match mode only checks for the presence of any match (match mode *first*). Additionally, this value can be minus one if the node was never evaluated, for example because it is part of an *or* expression. Only if the *count* modifier is used together with the substructure query operator, or the substructure operator is the range operator, the possibility of multiple matches is evaluated and larger values can be obtained. For these operations the default match mode is *distinctinneratoms* (see **`match ss`** command).

- *matchmask*
  A bitvector indicating which children of the root query node have matched. The length of the bitvector is the same as the number of children of the root node. A typical application of this retrieval item is in combination with a *range* node as root node.

- *maxscore*
  The maximum value of all computed scores, such as Cosine, Tanimoto or Tversky similarity scores, in the matching query for this result.

- *merit*
  For queries which use a merit/demerit rating scheme (for example, Bruns/Watson queries) this retrieves the accumulated merit/demerit sum of the top-level query node. The query needs to match for this retrieval to work, so in case none of the demerit rules match, you get an empty result, not a default zero merit/demerit value. Internally, there is no distinction between merit and demerit scores. The keyword *demerit* is an alias for this pseudo-property.

- *minscore*
  The minimum value of all computed scores, such as Cosine, Tanimoto or Tversky similarity scores, in the matching query for this result.

- *parent*
  The parent structure of the matching structure as a packed, base64-encoded serialized object string. If the dataset ensemble does not already contain it, it is computed from the structure as property `E_PARENT_STRUCTURE`.

- *productmatchatoms*
  The same as the *matchatoms* pseudo property, but for the ensemble on the right side of a matching reaction, not a simple structure. If no reaction was matched, this is an empty list.

- *productmatchbondatoms*
  The same as the *matchbondatoms* pseudo property, but for the ensemble on the right side of a matching reaction, not a simple structure. If no reaction was matched, this is an empty list.

- *productmatchbonds*
  The same as the *matchbonds* pseudo property, but for the ensemble on the right side of a matching reaction, not a simple structure. If no reaction was matched, this is an empty list.

- *queryid*
  The ID of the search tree query item which was responsible for the principal match. Every tree element of a query expression possesses an ID, starting with 1, and then assigned in incremental sequence from left to right in depth-first manner. For simple property or structure match expressions, the query ID is the ID of the matching branch, i.e. one for single-node expressions. For logical expressions with an *or*, *orcontinue* or *not* node, the overall reported query ID is that of the first matching leaf node. For expressions, where all leaves need to be checked, the query ID is the ID of the *and* or *eor* node where all leaves matched, not the ID of any individual leaf node.

- *reagentmatchatoms*
  The same as the *matchatoms* pseudo property, but for the ensemble on the left side of a matching reaction, not a simple structure. If no reaction was matched, this is an empty list.

- *reagentmatchbondatoms*
  The same as the *matchbondatoms* pseudo property, but for the ensemble on the left side of a matching reaction, not a simple structure. If no reaction was matched, this is an empty list.

- *reagentmatchbonds*
  The same as the *matchbonds* pseudo property, but for the ensemble on the left side of a matching reaction, not a simple structure. If no reaction was matched, this is an empty list.

- *record*
  The record number. In the context of in-memory datasets, this is the dataset list index of the matching object plus one. *rc* is an alias for this pseudo property.

- *rgatoms(rg)*
  A list of the atom labels in a matching structure which were mapped to an expanded R-group atom in the query. The property index is the name of the R-group of interest defined in the substructure, usually something like *R1*. If there was no expanded R-group of that name, the result list is empty.

- *rgattachments(rg)*
  A nested list of the atom label pairs of the bonds in a matching structure which connect between the structure framework and the atoms expanded as the named R-group *rg*. If there was no expanded R-group of that name, the result list is empty.
- *score*
  The first element of the *scores* array, as described below. If the query does not contain any scoring expressions, the result is empty.
- *scores*
  An integer vector of the results of all query expression branches, in depth-first left-to-right order, which computed a score, such as structure similarity queries with Cosine, Tanimoto or Tversky bitvector comparisons. In case a branch was not executed when the match was determined, a zero value is reported.
- *structure*
  The matched structure as a packed, base64-encoded serialized object string.
- *vrecord*
  For network object scans, this is always the same as *record*.

These pseudo properties are identical to those available for structure file queries. However, structure file queries allow the use of a couple of additional pseudo properties which are not supported for network queries.

network set

```
network set nhandle ?property value?...
network set nhandle ?dictionary?
n.set(property,value,...)
n.set({property:value,...})
n.property = value
n[property] = value
```

Standard data manipulation command. It is explained in more detail in the section about setting property data.

network setparam

```
network setparam nhandle property ?key value?...
network setparam nhandle property dictionary
n.setparam(property,?key,value?...)
n.setparam(property,dict)
```

Set or update a property computation parameter in the metadata parameter list of a valid property. This command is described in the section about retrieving property data. The current settings of the computation parameters in the property definition are not changed.

The return value is the updated property computation parameter dictionary.

network show

```
network show nhandle propertylist ?filterset? ?parameterdict?
n.show(property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **`network get`** command. The difference between **`network get`** and **`network show`** is that the latter does not attempt computation of property data, but raises an error if the data is not present and valid. For data already present, **`network get`** and **`network show`** are equivalent.

network sqldget

```
network sqldget nhandle propertylist ?filterset? ?parameterdict?
n.sqldget(property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **`network get`** command. The differences between **`network get`** and **`network sqldget`** are that the latter does not attempt computation of property data, but initializes the property value to the default and returns that default, if the data is not present and valid; and that the **SQL** command variant formats the data as **SQL** values rather than for **TCL** or **PYTHON** script processing.

network sqlget

```
network sqlget nhandle propertylist ?filterset? ?parameterdict?
n.sqlget(property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **`network get`** command. The difference between **`network get`** and **`network sqlget`** is that the **SQL** command variant formats the data as **SQL** values rather than for **TCL** or **PYTHON** script processing.

network sqlnew

```
network sqlnew nhandle propertylist ?filterset? ?parameterdict?
n.sqlnew(property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **`network get`** command. The differences between **`network get`** and **`network sqlnew`** are that the latter forces re-computation of the property data, and that the **SQL** command variant formats the data as **SQL** values rather than for **TCL** or **PYTHON** script processing.

network sqlshow

```
network sqlshow nhandle propertylist ?filterset? ?parameterdict?
n.sqlshow(property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **`network get`** command. The differences between **`network get`** and **`network sqlshow`** are that the latter does not attempt computation of property data, but raises an error if the data is not present and valid, and that the **SQL** command variant formats the data as **SQL** values rather than for **TCL** or **PYTHON** script processing.

network subcommands

```
network subcommands
dir(Network)
```

Lists all subcommands of the *network* command. Note that this command does not require a network handle.

network transfer

```
network transfer nhandle propertylist ?targethandle? ?targetpropertylist?
n.transfer(properties=,?target=?,?targetproperties=?)
```

Copy property data from one network to another network or other major object, without going through an intermediate scripting language object representation, or dissociate property data from the network. If a property in the argument property list is not already valid on the source network, an attempt is made to compute it.

If a target object is specified, the return value is the handle or reference of the target object. The source and target object cannot be the same object. In case a property associated with network minor objects (connections and vertices), the behavior is the same as described for ensemble minor objects in the documentation of **`ens transfer`**.

If a target property list is given, the data from the source is stored as content of a different property on the target. For this, the data types of the properties must be compatible, and the object class of the target property that of the target object. No attempt is made to convert data of mismatched types. In case of multiple properties, the source property list and the target property list are stepped through in parallel. If there is no target property list, or it is shorter than the source list, unmatched entries are stored as original property values, and this implies that the object class of the source and target objects are the same.

If no target object is specified, or it is spelled as an empty string or **PYTHON `None`**, the visible effect of the command is the same as a simple **`network get`**, i.e. the result is the property data value or value list. The property data is then deleted from the source object. In case the data type of the deleted property was a major object (i.e. an ensemble, reaction, table, dataset or network), it is only unlinked from the source object, but not destroyed. This means that the object handles returned by the command can henceforth the used as independent objects. They can be deleted by a normal object deletion command, and are no longer managed by the source object.

network unlock

```
network unlock nhandle propertylist/objclass/all
n.unlock(property=)
```

Unlock property data for the network, meaning that they are again under the control of the standard data consistency manager.

The property data to unlock can be selected by providing a list of the following identifiers:

- Property names or references
  Valid property instances on the network, or network minor objects are unlocked.
  Non-existent data is silently ignored. It is not possible to unlock individual property fields.
- *all*
  All valid network or network minor objects properties are unlocked.

- *vertex, connection, network*
  These are object class identifiers. All property data which is controlled by the network major object and attached to the specified object class is unlocked.

Property data locks are obtained by the **`network lock`** command.

The command returns the original network handle or reference.

network unpack

```
network unpack packstring ?compressionlib?
Network.Unpack(data=,?compressionlib=?)
```

Unpack a base64-encoded serialized object string which was created by a **`network pack`** command. The return value of this function is the handle or reference of the newly created network object, which is an exact duplicate of the packed original network.

Packed networks may also be unpacked by the **`network create`** command.

The default compression library is *zlib*. For more options, see **`network pack`**.

Example:

```
set packdata [network pack $nhandle]
set nhandle [network unpack $packdata]
```

network valid

```
network valid nhandle propertylist
n.valid(property/propertysequence)
```

Returns a list of boolean values indicating whether values for the named properties are currently set for the network or its minor objects. No attempt at computation is made.

Example:

```
network valid $nhandle V_ONTOLOGY_TERM
```

will report whether the network has ontology term definitions attached to its vertices or not.

`network has` is an alias to this command.

network verify

```
network verify nhandle property
n.verify(property)
```

Verify the values of the specified property on the network. The property data must be valid, and a network or network minor sub-object property. If the data can be found, it is checked against all constraints defined for the property, and, if such a function has been defined, is tested with the value verification function of the property.

If all tests are passed, the return value is boolean 1, 0 if the data could be found but fails the tests, and an error condition otherwise.

network vertices

```
network vertices nhandle ?filterset? ?filtermode?
n.vertices(?filters=?,?mode=?)
```

Standard cross-referencing command to obtain the labels or references of the vertices the network contains as minor objects. This is explained in more detail in the section about object cross-references.

Examples:

```
network vertices $nhandle
filter create rootnode property V_LEVEL value 0 operator =
network vertices $nhandle rootnode
```

The example simply returns a list of the labels of the vertices the network contains as minor objects. The second example restricts these to the subset where property `V_LEVEL` has a specific value.

network write

```
network write nhandle filename ?format?
n.write(filename=,?format=?)
```

Write the contents of a network object to a file. The output file format is deduced from the suffix of the file name. If it is not recognized, the native toolkit format (default suffix *.nbin)* is used. This default mechanism can be overridden by giving an explicit format name in the optional argument.

The currently supported network output formats are **Cactvs** native (*.nbin*), **SIF** (*.sif*), **GML** (*.gml*), **XGMML** (*.xgmml*, *.gr*), **KNIME** network data (*.knet*) , **BEEF** (*.beef*)., **GraphViz** (*.gv*), and **KEGG** metabolic network (*.kgml*). **OBO** and **GeneOntology** ontology files can only be read, but not written. Predefined format alias names include *knime* for **KNET** files, and *bisonet* for **BEEF**. Zipped **BEEF** files cannot be written directly at this time, but can readily be obtained by first writing an unpacked *BEEF* file and then running an external standard zip-compatible compressor on it.

The file name may be either a disk file name, one of the magic file names *stdout* or *stderr*, a pipe construct, or, on Unix-class operating systems, an open **Tcl** or **Python** file or socket handle.

The command returns the original network handle or reference.

The *pi* Command

The *pi* command is the generic command used to manipulate π systems. The syntax of this command follows the standard schema of *command/subcommand/majorhandle/minorlabel*.

Pseudo π system labels *first*, *last* and *random* are special values, which select the first π system in the π system list, the last, or a random π system.

Examples:

```
pi get $ehandle 1 P_ATOMS
```

This is the list of officially supported subcommands:

pi append

```
pi append ehandle label ?property value?...
p.append({?property:value,?...})
p.append(?property,value,?...)
```

Standard data manipulation command for appending property data. It is explained in more detail in the section about setting property data.

Example:

```
pi append $ehandle 1 P_NAME “_uvactive”
```

pi atoms

```
pi atoms ehandle label ?filterset? ?filtermode?
p.atoms(?filters=?,?mode=?)
```

Standard cross-referencing command to obtain the labels or references of the atom in the π system. This is explained in more detail in the section about object cross-references.

Example:

```
pi atoms $ehandle 1 carbon
```

returns the labels of the carbon atoms in the π system.

pi bonds

```
pi bonds ehandle label ?filterset? ?filtermode?
p.bonds(?filters=?,?mode=?)
```

Standard cross-referencing command to obtain the labels or references of the bonds the π system contains. This is explained in more detail in the section about object cross-references. Technically, a π system contains atoms, not bonds. This command lists all bonds which exist between atoms in the π system. Bonds involving only a single atom in a π system are excluded.

Examples:

```
pi bonds $ehandle 1
pi bonds $ehandle 1 {1 doublebond triplebond} count
```

The first example returns all labels of the bonds π system 1 contains. The second example returns the number of double or triple bonds in the π system.

pi create

```
pi create ehandle ?atom/atomlist?...
Pi(eref,?aref/arefsequence/alabel?,...)
Pi(aref,...)
Pi.Create(eref,?aref/arefsequence/alabel?,...)
Pi.Create(aref,...)
```

Define a new π system from an atom set. A new π system is always created, even if one with the same atoms already exists. No check on the presence of π electrons is performed. Before the command is executed, the default π system set is automatically instantiated if it was not yet computed. Adding a new π system invalidates properties which are sensitive to π system changes.

The command returns the label of the new π system.

pi defined

```
pi defined ehandle label property
p.defined(property)
```

This command checks whether a property is defined for the π system. This is explained in more detail in the section about property validity checking. Note that this is *not* a check for the presence of property data! The **`ens valid`** command is used for this purpose.

pi delete

```
pi delete ehandle ?label?...
pi delete ehandle all
p.delete()
Pi.Delete(eref,?pref/plabel/prefsequence?,...)
Pi.Delete(pref,...)
Pi.Delete(eref,"all")
```

This command removes π systems from the ensemble π system list and destroys them. A *pi* property invalidation event is generated and thus the command may indirectly change the ensemble data.

This command is rarely used. π systems are usually generated and destroyed automatically.

The command returns the number of deleted items.

pi dget

```
pi dget ehandle label propertylist ?filterset? ?parameterdict?
p.dget(property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **`pi get`** command. The difference between **`pi get`** and **`pi dget`** is that the latter does not attempt computation of property data, but rather initializes the property values to the default and returns that default if the data is not yet available. For data already present, **`pi get`** and **`pi dget`** are equivalent.

pi ens

```
p.ens()
```

PYTHON-only method to get the ensemble reference from a π system reference.

pi exists

```
pi exists ehandle label ?filterlist?
p.exists(?filters=?)
Pi.Exists(eref,label,?filters=?)
```

Check whether this π system exists. Optionally, a filter list can be supplied to check for the presence of specific features. The command returns boolean 0 if the π system does not exist, or fails the filter, and 1 in case of successful testing.

Example:

```
pi exists $ehandle 99
```

pi expr

```
pi expr ehandle label expression
p.expr(expression)
```

Compute a standard **SQL**-style property expression for the π system. This is explained in detail in the chapter on property expressions.

pi fill

```
pi fill ehandle label ?property value?...
p.fill({property:value,...})
p.fill(?property,value?,...)
```

Standard data manipulation command for setting data, ignoring possible mismatches between the lengths of the lists of objects associated with the property and the value list. It is explained in more detail in the section about setting property data.

pi filter

```
pi filter ehandle label filterlist
p.filter(filters)
```

Check whether a π system passes a filter list. The return value is boolean 1 for success and 0 for failure.

Example:

```
pi filter $ehandle 1 [list carbon doublebond]
```

checks whether the π system contains one or more carbon atoms and one or more double bonds. The double bond does not need to contain a carbon atom.

pi get

```
pi get ehandle label propertylist ?filterset? ?parameterdict?
p.get(property=,?filters=?,?parameters=?)
p[property]
p.property
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

pi groups

```
pi groups ehandle label ?filterset? ?filtermode?
```

```
p.groups(?filters=?,?mode=?)
```

Standard cross-referencing command to obtain the labels or references of the groups the π system overlaps with. This is explained in more detail in the section about object cross-references. An overlap between a π system and a group is established when there are common atoms which are contained in both objects.

Example:

```
pi groups $ehandle 1
```

pi index

```
pi index ehandle label
p.index()
```

Get the index of the π system. The index is the position in the π system list of the ensemble. The first position is index 0.

Example:

```
pi index $ehandle 99
```

pi jget

```
pi jget ehandle label propertylist ?filterset? ?parameterdict?
p.jget(property=,?filters=?,?parameters=?)
```

This is a variant of **`pi get`** which returns the result data as a **JSON** formatted string instead of **TCL** or **PYTHON** interpreter objects.

pi jnew

```
pi jnew ehandle label propertylist ?filterset? ?parameterdict?
p.jnew(property=,?filters=?,?parameters=?)
```

This is a variant of **`pi new`** which returns the result data as a **JSON** formatted string instead of **TCL** or **PYTHON** interpreter objects.

pi jshow

```
pi jshow ehandle label propertylist ?filterset? ?parameterdict?
p.jshow(property=,?filters=?,?parameters=?)
```

This is a variant of **`pi show`** which returns the result data as a **JSON** formatted string instead of **TCL** or **PYTHON** interpreter objects.

pi local

```
pi local ehandle label propertylist ?filterset? ?parameterdict?
p.local(property=,?filters=?,?parameters=?
```

Standard data manipulation command for reading and recalculating object data. It is explained in more detail in the section about retrieving property data.

pi match

```
pi match ehandle label ss_ehandle ?ss_label? ?matchflags? ?ignoreflags?
   ?atommatchvar? ?bondmatchvar? ?molmatchvar?
p.match(substructure=,?substructurepisystem=?,?matchflags=?,?ignoreflags=?,
   ?atommatchvariable=?,?bondmatchvariable=?,?molmatchvariable=?)
```

Check whether the selected π system matches a substructure. Only the first substructure π system, or the π system selected by the substructure label parameter, is tested. The substructure may be part of any structure ensemble, and even be in the same ensemble as the primary command π system. Both the atoms in the π system and the bonds between them are checked.

The precise operation of the substructure match routine can be tuned by providing a standard set of match flags and feature ignore flags. The default match flag set has set bits for the *bondorder*, *atomtree* and *bondtree* comparison features, and an empty ignore set. If a flag set is specified as an empty string, the default set is used. In order to reset a flag set, an explicit *none* value must be used.

The command returns 1 for a successful match, 0 otherwise. If an optional atom, bond, or molecule match variable is specified, it is set to a nested list of matching substructure/structure atom, bond or molecule labels. If no match can be found, the variable is set to an empty list, or **None** for **Python**. In case only a bond or molecule match variable is needed, an empty string can be used to skip the unused match variable argument positions.

Examples:

```
pi match [ens create C=CCC=N] 1 C=C
pi match [ens create C=CCC=N] 1 N=C
pi match [ens create C=CCC=N] 2 C=C
pi match [ens create C=CCC=N] 2 N=C
```

The first and last commands match, the second and third do not.

pi mol

```
pi mol ehandle label ?filterset? ?filtermode?
p.mol(?filters=?,?mode=?)
```

Standard cross-referencing command to obtain the labels or references of the molecules the π system is contained in. This is explained in more detail in the section about object cross-references.

pi new

```
pi new ehandle label propertylist ?filterset? ?parameterdict?
p.new(property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

The difference between *pi get* and *pi new* is that the latter forces the re-computation of the property data, regardless whether it is present and valid, or not.

pi nget

```
pi nget ehandle label propertylist ?filterset? ?parameterdict?
p.nget(property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **`pi get`** command. The difference between **`pi get`** and **`pi nget`** is that the latter returns numeric data, even if symbolic names for the values are available.

pi pi

```
pi pi ehandle label
Pi.Ref(eref,identifier)
```

Standard cross-referencing command to obtain the label or reference of the π system as stored in property `P_LABEL`. This is explained in more detail in the section about object cross-references.

Example:

```
pi pi $ehandle #0
```

returns the label of the first π system of the ensemble π system list.

pi ref

```
Pi.Ref(eref,identifier)
```

PYTHON only method to get a π system reference. See **`pi pi`** command.

pi rings

```
pi rings ehandle label ?filterset? ?filtermode?
p.rings(?filters=?,?mode=?)
```

Standard cross-referencing command to obtain the labels or references of the rings the π system is associated with. This is explained in more detail in the section about object cross-references. Rings which only partially overlap with the π system are included.

Examples:

```
pi rings $ehandle 1
pi rings $ehandle 1 [list heterocycle aroring]
```

The first example returns the labels or references of all rings the π system overlaps with. If the π system does not overlap with any ring, an empty list is returned. Only labels of rings in the SSSR or **ESSSR** set are returned, even if the currently computed ring set is larger. The second example filters the rings - only heteroaromatic rings are reported.

pi ringsystems

```
pi ringsystems ehandle label ?filterset? ?filtermode?
p.ringsystems(?filters=?,?mode=?)
```

Standard cross-referencing command to obtain the labels or references of the ring systems the π system is associated with. This is explained in more detail in the section about object cross-references. Ring systems which only partially overlap with the π system are included.

pi set

```
pi set ehandle label ?property value?...
p.set(?property,value?,...)
p.set({property:value,...})
p.property = value
p[property] = value
```

Standard data manipulation command. It is explained in more detail in the section about setting property data.

pi show

```
pi show ehandle label propertylist ?filterset? ?parameterdict?
p.show(property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the *pi get* command. The difference between **`pi get`** and **`pi show`** is that the latter does not attempt computation of property data, but raises an error if the data is not present and valid. For data already present, **`pi get`** and **`pi show`** are equivalent.

pi sigmas

```
pi sigmas ehandle label ?filterset? ?filtermode?
p.sigmas(?filters=?,?mode=?)
```

Standard cross-referencing command to obtain the labels or references of the σ systems the π system overlaps with. This is explained in more detail in the section about object cross-references.

Examples:

```
pi sigmas $ehandle 1
```

σ systems are a rather exotic feature and not commonly used. These are essentially descriptions of bonding interactions which use s orbitals, such as normal, covalent single bonds, or the central bond in multiple bonds. A simple double bond is described with one σ system and one π system in this representation.

pi sqldget

```
pi sqldget ehandle label propertylist ?filterset? ?parameterdict?
p.sqldget(property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **`pi get`** command. The differences between **`pi get`** and **`pi sqldget`** are that the latter does not attempt computation of property data, but initializes the property value to the default and returns that default, if the data is not present and valid; and that the **SQL** command variant formats the data as **SQL** values rather than for **TCL** or **PYTHON** script processing.

pi sqlget

```
pi sqlget ehandle label propertylist ?filterset? ?parameterdict?
p.sqlget(property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **`pi get`** command. The difference between **`pi get`** and **`pi sqlget`** is that the **SQL** command variant formats the data as **SQL** values rather than for **TCL** or **PYTHON** script processing.

pi sqlnew

```
pi sqlnew ehandle label propertylist ?filterset? ?parameterdict?
p.sqlnew(property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **`pi get`** command. The differences between **`pi get`** and **`pi sqlnew`** are that the latter forces re-computation of the property data, and that the **SQL** command variant formats the data as **SQL** values rather than for **Tcl** or **Python** script processing.

pi sqlshow

```
pi sqlshow ehandle label propertylist ?filterset? ?parameterdict?
p.sqlshow(property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **`pi get`** command. The differences between **`pi get`** and **`pi sqlshow`** are that the latter does not attempt computation of property data, but raises an error if the data is not present and valid, and that the **SQL** command variant formats the data as **SQL** values rather than for **Tcl** or **Python** script processing.

pi subcommands

```
pi subcommands
dir(Pi)
```

Lists all subcommands of the **`pi`** command. Note that this command does not require an ensemble handle, or a label.

pi surfaces

```
pi surfaces ehandle label ?filterset? ?filtermode?
p.surfaces(?filters=?,?mode=?)
```

Standard cross-referencing command to obtain the labels or references of surface patches the π system is associated with. This is explained in more detail in the section about object cross-references.

Example:

```
pi surfaces $ehandle $label
```

Note that surface patches do not need to be associated with an atom, and if they are not, they are implicitly not associated with any π system.

pi xbonds

```
pi xbonds ehandle label ?filterset? ?filtermode?
p.xbonds(?filters=?,?mode=?)
```

Get labels or references of crossing bonds which are not contained in the π system (defined as all bonds between the registered atoms in the system), but have one atom in the π system.

The query Command

The *query* command creates a reusable object from the query expression parameter of a scan command (**molfile scan**, **dataset scan**, etc.). All commands where such a query expression can be used as a string parameter alternatively accept the handle or reference of a query object in its place.

Additionally, query objects offer method to translate the query from the native toolkit syntax into other formats, such as **SQL**.

Examples:

```
set qh [query create "structure >= c1ncccc1"]
set n [molfile scan $fh $qh count]
```

This is the list of currently officially supported subcommands:

query create

```
query create ?querystring?
Query(?querystring)?
Query.Create(?querystring?)
```

Create a new query object from a query expression entered as a string. It is possible to omit the query string or pass en empty string or **Null**. In that case, a query which always matches is created.

Example:

```
set qh [query create {and {E_NMOLECULES = 1} {structure >= c1ncccc1}}]
```

query delete

```
query delete all
query delete ?qhandlelist?...
q.delete()
Query.Delete("all")
Query.Delete(?qrefsequence/qref/qhandle?,...)
```

Delete query objects. The special parameter *all* may be used to delete all queries currently registered in the application. Alternatively, any number of lists of query handles may be specified for specific query deletions.

The command returns the number of deleted queries.

Example:

```
query delete $qhandle
```

query get

```
query get qhandle attribute
q.get(attribute)
q.attribute
q[attribute]
```

Get an attribute value of the query objects. Besides the settable attributes listed under **query set**, the following attributes are read-only:

- *configuration*
  A dictionary of the currently configured values for query properties, fields and table names.
- *sql_query*
  The query recoded as an **SQL** query on a simple table schema, with referenced table and column names as defined in the query object configuration.

query list

```
query list ?pattern?
Query.List(?pattern=?)
```

Return a list of the handles of all currently existing query objects in the application. If desired, the list can be filtered by a string pattern.

query print

```
query print qhandle
q.print()
```

Return the string representation of the query in native toolkit format.

query ref

```
Query.Ref(identifier)
```

PYTHON only method to get a query reference from a handle or another identifier.

query set

```
query set qhandle ?attribute value?...
query set qhandle attributedict
q.set(attributedict)
q.set(attribute,value,...)
q.att = value
q[att] = value
```

Configure the query object. The following attributes can be set and also read via `query get`:

- *atomtable*
  Set the name of the atom table for translating the query into **SQL**.
- *bondtable*
  Set the name of the bond table for translating the query into **SQL**.
- *ens_elementcount_field*
  Set the name of the column with an array of element counts in the ensemble table for translating the query into **SQL**.
- *ens_elementcount_property*
  Set the property used to compute ensemble element counts for formula searching.
- *ens_isotopehash_field*
  The name of the column with isotope-aware ensemble hashes in the ensemble table for translating the query into **SQL**.

- *ens_isotopehash_property*
  The property used to compute isotope-aware ensemble hashes. This is by default inherited from the control variable **`cactvs(default_structure_isotope_hash_property)`**.
- *ens_isotopestereohash_field*
  Set the name of the column with isotope- and stereo-aware ensemble hashes in the reaction table for translating the query into **`SQL`**.
- *ens_isotopestereohash_property*
  The property used to compute isotope- and stereo-aware ensemble hashes. This is by default inherited from the control variable
  `cactvs(default_structure_isotope_stereo_hash_property)`.
- *ens_similarityscreen_field*
  The name of the column with the similarity screen bitvector.
- *ens_similarityscreen_property*
  The property used to compute the similarity screen bitvector. This is by default inherited from the control variable **`cactvs(default_similarity_property)`**.
- *ens_simplehash_field*
  The name of the column with simple ensemble hashes in the ensemble table for translating the query into **`SQL`**.
- *ens_simplehash_property*
  Set the property used to compute simple ensemble hashes. This is by default inherited from the control variable **`cactvs(default_structure_simple_hash_property)`**.
- *ens_simpletautohash_field*
  Set the name of the column with simple tautomer-tolerant ensemble hashes in the ensemble table for translating the query into **`SQL`**.
- *ens_simpletautohash_property*
  The property used to compute simple tautomer-tolerant ensemble hashes. This is by default inherited from the control variable **`cactvs(default_structure_tauto_hash_property)`**.
- *ens_stereohash_field*
  The name of the column with stereo-aware ensemble hashes in the ensemble table for translating the query into **`SQL`**.
- *ens_stereohash_property*
  Set the property used to compute stereo-aware ensemble hashes. This is by default inherited from the control variable **`cactvs(default_structure_stereo_hash_property)`**.
- *ens_substructurescreen_field*
  The name of the column with the substructure screen bitvector
- *ens_substructurescreen_property*
  The property used to compute the substructure screen bitvector. This is by default inherited from the control variable **`cactvs(default_substructure_screen_property)`**.
- *ens_superstructurescreen_field*
  The name of the column with the superstructure screen bitvector

- *ens_superstructurescreen_property*
  The property used to compute the superstructure screen bitvector. This is by default inherited from the control variable
  `cactvs(default_superstructure_screen_property)`.
- *enstable*
  The name of the base ensemble-level table for translating the query into **SQL**.
- *fileassociation*
  The handle or reference of an open structure or reaction data file. Some file formats contain metadata on the internal configuration of the files, which are then automatically used to configure the right properties and field names for standard structure and reaction queries on this file. Providing a file reference is not required, it can just make the query configuration set-up easier.
- *flags*
  A bit-ored combination of various flags. The currently supported flags are *ignorethresholds* and *screeningonly.*
- *keycolumn*
  The name of tghe **SQL** column which links ensembles and atom/bond tables.
- *querystring*
  The string form of the query. Setting this re-configures the query object to check a different match condition.
- *reaction_isotopehash_field*
  Set the name of the column with isotope-aware reaction hashes in the reaction table for translating the query into **SQL**.
- *reaction_isotopehash_property*
  Set the property used to compute isotope-aware reaction hashes. This is by default inherited from the control variable **`cactvs(default_reaction_isotope_hash_property)`**.
- *reaction_isotopestereohash_field*
  Set the name of the column with isotope- and stereo-aware reaction hashes in the reaction table for translating the query into **SQL**.
- *reaction_isotopestereohash_property*
  Set the property used to compute isotope- and stereo-aware reaction hashes. This is by default inherited from the control variable
  `cactvs(default_reaction_isotope_stereo_hash_property)`.
- *reaction_screen_field*
  The name of the column with the reaction screen bitvector
- *reaction_screen_property*
  The property used to compute the reaction screen bitvector. This is by default inherited from the control variable **`cactvs(default_reaction_screen_property)`**.
- *reaction_simplehash_field*
  Set the name of the column with simple reaction hashes in the reaction table for translating the query into **SQL**.

- *reaction_simplehash_property*
  The property used to compute simple reaction hashes. This is by default inherited from the control variable **`cactvs(default_reaction_simple_hash_property)`**.
- *reaction_stereohash_field*
  Set the name of the column with stereo-aware reaction hashes in the reaction table for translating the query into **SQL**.
- *reaction_stereohash_property*
  The property used to compute stereo-aware reaction hashes. This is by default inherited from the control variable **`cactvs(default_reaction_stereo_hash_property)`**.
- *reactiontable*
  The name of the base reaction-level table for translating the query into **SQL**.
- *sql_dialect*
  The **SQL** dialect to use when translating the query to **SQL**. This is by default inherited from the **`cactvs(sql_dialect)`** control variable.

query subcommands

```
query subcommands
dir(Query)
```

Lists all subcommands of the **`query`** command. Note that this command does not require a query handle.

The *reaction* Command

The *reaction* command is the generic command used to manipulate reactions. The syntax of this command follows the standard schema of *command/subcommand/majorhandle*. Reactions are major objects and thus do not need any minor object labels for identification.

Examples:

```
reaction get $xhandle X_IDENT
```

This is the list of currently officially supported subcommands:

reaction add

```
reaction add xhandle ?ReactionSMILES/SMIRKS/BASE64Blob ?decodermode??
reaction add xhandle ?ehandle?...
reaction add xhandle ?{ehandle role}?...
x.add(?ReactionSMILES/SMIRKS/BASE64Blob?,?decodermode?)
x.add(eref,...)
x.add((eref,role),...)
```

Add ensembles to a reaction. The syntax of the various variants to specify ensembles and their reaction roles are the same as in the **`reaction create`** command.

Example:

```
reaction add [reaction create] $reagent_ehandle $product_ehandle
```

reaction append

```
reaction append xhandle ?property value?...
x.append({?property:value,?...})
x.append(?property,value,?...)
```

Standard data manipulation command for appending property data. It is explained in more detail in the section about setting property data.

The command returns the first data value.

Example:

```
reaction append $xhandle X_NAME "_new"
```

reaction assign

```
reaction assign xhandle srcproperty dstproperty
x.assign(srcproperty=,dstproperty=)
```

Assign property data to another property on the same reaction. Both properties must be associated with the reaction object class. This process is more efficient than going through a pair of **`reaction get/reaction set`** commands, because in most cases no string or TCL/PYTHON script object representations of the property data need to be created.

Both source and destination properties may be addressed with field specifications. A data conversion path must exist between the data types of the involved properties. If any data conversion fails, the command fails. For example, it is possible to assign a string property to a numeric property - but only if all property values can be successfully converted to that numeric type. The reverse example case always succeeds, out-of-memory errors and similar global events excluded.

The original property data remains valid. The command variant **`reaction rename`** directly exchanges the property name without any data duplication or conversion, if that is possible. In any case, the original property data is no longer present after the execution of this command variant.

If the properties are not associated with reactions (prefix `X_`), the operation is performed on all reaction ensembles.

The command returns the original object handle for **Tcl**, or object reference for **Python**.

Examples:

```
reaction assign $xh A_XY A_XY%
```

This code snippet creates a backup atomic 2D layout coordinates on all reaction ensembles.

```
reaction rename $xh X_IDENT X_NAME
```

Reassign the data in property `X_IDENT` to property `X_NAME`. If possible, this is done without memory reallocation and decoding/encoding procedures.

reaction cast

```
reaction cast xhandle dataset/ens/reaction/table ?propertylist?
x.cast(objectclass=,?properties=?)
```

Transform the reaction into a different object. Depending on the target object class, the result is as follows:

- *dataset*
  A new dataset which contains which contains the reaction as first object.
- *ens*
  The reagent ensemble of the reaction, or the first other ensemble in the reaction is there is no reagent, or a newly created empty ensemble as last resort. The rest of the reaction and reaction ensembles are destroyed.
- *reaction*
  Only supplied for the sake of completeness. This mode does nothing.
- *table*
  A new table with one row and automatically generated columns for all properties of the input reaction of the *reaction* (X_*) object class. The row is filled with the input reaction data, and the reaction is moved to the internal dataset of the table.

If the optional property list is specified, an attempt is made to compute the listed properties before the cast operation, so that they may become a part of the new object. No error is raised if a computation fails.

The command returns the handle or reference of the new object, or the input object handle or reference in case of mode *reaction*.

reaction clear

```
reaction clear xhandle ?role? ?deleteensembles?
x.clear(?role=?,?deleteensembles=?)
```

Remove and optionally delete ensembles of a reaction. By default, all reaction ensembles are moved out of the reaction, but they are not deleted. If a reaction role is given (possible roles are taken from the enumeration of property `E_REACTION_ROLE`, the default set is *unknown, reagent, product, solvent, catalyst, intermediate, impurity, byproduct, agent* and *waste*), only those ensembles with the specified role are removed. If the *deleteensemble* flag is set, targeted ensembles are not simply removed from reaction membership, but destroyed.

The command returns the count of removed or deleted ensembles.

Examples:

```
reaction clear $xhandle
reaction clear $xhandle solvent 1
```

The first example removes all ensembles from the reaction, but keeps them in memory, and they can still be accessed via their handles. The second example removes all solvent ensembles from the reaction and destroys them.

reaction copy

```
reaction copy src_xhandle dst_xhandle
x.copy(xref_dst)
```

Create a copy of the input reaction in the framework of an existing reaction. The old data of the destination reaction is destroyed, but its handle is reused for the copy. The destination handle can be an empty string. In that case, the reaction is duplicated and a new handle assigned.

This command is useful when references to a reaction handle are potentially stored in unknown locations, and the reaction needs to be updated.

The return value of the command is the handle or reference of destination reaction. It is allowed to copy a reaction onto itself.

reaction create

```
reaction create
   ?RxnSMILES/SMIRKS/RInChI/BASE64Blob/KEGGID/patran/RxnMinimol/HexRxnMinimol?
   ?decodermode?
reaction create ?ehandle?...
reaction create ?{ehandle role}?...
Reaction(?RxnSMILES/SMIRKS/RInchI/BASE64Blob/KEGGID/patran/RxnMinimol/HexRxnMini
mol?,?decodermode?)
Reaction(eref,...)
Reaction((eref,role),...)
Reaction.Create(RxnSMILES/SMIRKS/RInchI/BASE64Blob/KEGGID/patran/RxnMinimol/HexR
xnMinimol,?decodermode?)
Reaction.Create(eref,...)
Reaction.Create((eref,role),...)
```

This command creates a new reaction. Without any parameters, an empty reaction without any ensembles in it is made. The return value is the new reaction handle or reference.

Example:

```
set xhandle [reaction create]
```

The first command variant is intended for creating a reaction with data is the use of a single-argument line notation. The supported line notations include Reaction **SMILES** or **SMIRKS**, **RInChI** strings, hex-encoded versions thereof, a **KEGG** reaction identifier in the form *RPxxxxx*, a **Cactvs** base64-encoded serialized reaction object string (see **`reaction pack`**), a raw reaction

Minimol byte blob (computed as property X_MINIMOL), a hex-encoded form thereof, or a base64-encoded compressed file content, such as an **MDL RXN** file record. For the last variant, the compression algorithm may be raw *zlib, gzip* or *zip* and is automatically detected. Additionally, any of those forms may be passed as a data **URI**. If a data **URI** is detected, its payload is extracted and used as argument in a second pass. The data for the decoding of **KEGG** IDs is downloaded from the **KEGG** site via an **HTTP** connection and requires that the interpreter is allowed port 80 Internet access.

Similar to the **ens create** command, it is also possible to prefix the structure encoding, if it is a line notation or an encoding without line breaks, with *smiles: smirks: rinchi:* or *kegg:* in order to explicitly name the encoding of these formats.

The only multi-line encoding recognized by this command are **Lhasa Patran** reaction patterns. They are automatically decoded in a form which makes them suitable for use in *lhasa* reaction processor objects. Reactions stored in uncompressed files must be read by means of a **molfile read** command.

Without setting a specific decoder mode in the following optional argument, the data string is decoded in the format-specific default mode (i.e. as standard **SMILES** strings for the **SMILES** family) and molecules with a complete hydrogen set are generated if the encoding supports such a distinction.

For **Reaction SMILES** variants, the explicit decoder modes *smarts* (alias *smirks*)*, strictsmarts* (alias *strictsmirks*)*, hadd, nohadd, transform* or *pattern* (alias *patran*) can be requested. In *smarts* or *strictsmarts* mode, full **SMARTS** or **SMIRKS** encodings are recognized, which are useful for reaction substructure searches or **SMIRKS** transforms. The difference between *smarts* and *strictsmarts* is that the latter enforces aliphaticity checks (see **ens create**). No implicit hydrogen is added in that mode. For special purposes, there is also a *strictatomsmirks*/*strictatomsmarts* mode, which enforces strict rules only for atom matching, but not bond matching. The *nohadd* mode does not supports **SMARTS** constructs, but neither are implicit hydrogens are instantiated.The *hadd* mode adds a standard hydrogen set. In order to force a full hydrogen addition to the raw decoded reaction even if it would not be done otherwise, use the mode *forcehadd*. The *transform* mode is similar to the *smarts* mode, but additionally hydrogen atoms encoded explicitly are not instantiated but rather translated into a hydrogen count query specification. This is useful for some classes of **SMIRKS** transforms. **SMILES** atom mapping specifications are allowed in all modes. The explicit *pattern* or *patran* modes should be used when decoding *patran* patterns - but then this is already the default for this type of data.

In case the argument is a **Reaction SMILES** variant or an **RInChI** string, and an *agent* in Daylight nomenclature is specified (the middle section between the >> characters), it is assigned the reaction role *agent*, which is rather unspecific. It may subsequently be changed by setting the E_REACTION_ROLE property of the agent ensemble.

If the argument of the command is a single ensemble handle or reference, and this ensemble possesses valid A_CGR_CHANGE and B_CGR_CHANGE properties indicating that it is a condensed graph representation of a reaction (property X_CGR), the **CGR** representation is expanded and the new reaction consists of two newly created reagent and product ensembles. The original input structure remains unchanged, and does not become a reaction member ensemble.

Ensembles which are decoded from the arguments have normal object handles or references and may be addressed via these just like ensembles which are not part of a reaction.

Examples:

```
set xhandle [reaction create C=O>Cc1ccccc1>CO]
```

```
ens set [reaction ens $xhandle agent] E_REACTION_ROLE solvent
set xhandle [reaction create {[C:1]=[O:2]>>[C:1][O:2]} smirks]
set xhandle [reaction create KEGG:RP09586]
set xhandle [reaction create [list C1CCCCC1 reagent] [list c1ccccc1 product]]
```

The second **`reaction create`** command variant uses a set of ensemble handles as arguments and makes these ensembles members of the newly created reaction. The reaction role of the ensembles is determined by default by their position in the argument list. The first ensemble is the *reagent*, followed by *product, solvent, catalyst, intermediate, impurity, byproduct, agent* and *waste* in that order. It is possible to skip a role by providing an empty string (or **`None`** in **PYTHON**) as argument placeholder. If necessary, ensembles are removed from an existing reaction and transferred to the new one, since an ensemble cannot be a member of more than one reaction at a time. In case the default order-dependent reaction role assignment is not convenient, any argument may be specified as a sequence of two components instead. In that notation, the first list element is the normal ensemble handle or reference and the second the applicable role designator encoded as a string from above set.

Examples:

```
set xhandle [reaction create $reagent_ehandle [ens create CO] {} {[Pt]}]
```

This example creates a reaction from an existing *reagent* ensemble and then adds a *product* ensemble and platinum as *catalyst*. The *solvent* parameter position is skipped by entering an empty string.

Finally, it is possible to explicitly specify the role by not just providing an ensemble handle as parameter, but a list consisting of an ensemble handle and its reaction role.

Example:

```
set xhandle [reaction create [list $product_handle product] \
   [list [ens create {[Pt]} catalyst]]
```

reaction dataset

```
reaction dataset xhandle ?filterlist?
x.dataset(?filters=?)
```

Return the dataset handle or reference of the dataset the reaction is a member of. It the reaction is not member of a dataset, or does not pass all of the optional filters, an empty string or **`None`** for **PYTHON** is returned.

Example:

```
reaction dataset $xhandle
```

reaction defined

```
reaction defined xhandle property
x.defined(property)
```

This command checks whether a property is defined for the reaction. This is explained in more detail in the section about property validity checking. Note that this is *not* a check for the presence of property data! The **`reaction valid`** command is used for this purpose.

reaction delete

```
reaction delete all
reaction delete ?xhandlelist?...
x.delete()
Reaction.Delete("all")
Reaction.Delete(?xrefsequence/xref/xhandle?,...)
```

Delete reactions and the ensembles which are part of the deleted reactions. The special parameter *all* may be used to delete all reactions currently registered in the application. Alternatively, any number of lists of reaction handles may be specified for specific reaction deletions. If a reaction is part of a linked reaction network, the reaction is removed from the network, but the network continues to exist as long as there are other linked reactions in it.

The command returns the number of deleted reactions.

Example:

```
reaction delete $xhandle
reaction delete $xhandlelist1 $xhandlelist2
```

reaction dget

```
reaction dget xhandle propertylist ?filterset? ?parameterdict?
x.dget(property=,?filters=?,?parameters=?)
Reaction.Dget(data,property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **`reaction get`** command. The difference between **`reaction get`** and **`reaction dget`** is that the latter does not attempt computation of property data, but rather initializes the property values to the default and return that default if the data is not yet available. For data already present, **`reaction get`** and **`reaction dget`** are equivalent.

The **PYTHON** class method is a one-shot command. The transient ensemble created from the initialization items is automatically deleted when the command finishes. The data for the creation of the temporary ensemble is equivalent to the first argument of the standard constructor. Additional constructor parameters cannot be used.

reaction dup

```
reaction dup xhandle ?dataset? ?position?
x.dup(?target=?,?position=?)
```

Duplicate the specified reaction and its ensembles. The return value is the new reaction handle or reference. The duplicated reaction ensembles are also assigned unique handles.

The duplicate ensemble is placed into the same dataset as the source, if it is a member of a dataset. Specifying an explicitly empty dataset argument (including **`None`** for **PYTHON**) places the duplicate outside any dataset, regardless of the dataset membership of the source ensemble.

If the duplicate is moved to a dataset, it is appended to the dataset end by default. This happens also if the position parameter is explicitly specified as *end* or an empty string. Otherwise, the reaction is inserted at the given position, starting with 0. If the requested position is larger than the current size of the dataset, the reaction is appended.

The **`reaction hdup`** command is a variant of this command. It automatically adds a hydrogen set to the duplicate.

If the reaction is part of a linked reaction network, the network is not duplicated. The new reaction is not linked to any other reactions. You can use the *xdup* command variant for this purpose.

Example:

```
set xdup [reaction dup $xhandle]
```

reaction ens

```
reaction ens xhandle ?filterlist?
x.ens(?filters=?)
```

Return a list of the handles of the ensembles which are a part of the reaction. Optionally, these ensembles may be filtered by a simple filter list.

Example:

```
reaction ens $xhandle metal
```

Find all ensembles in the reaction which contain one or more metal atoms.

reaction exists

```
reaction exists xhandle ?filterlist?
x.exists(?filters=?)
Reaction.Exists(xref=,?filters=?)
```

Check whether a reaction handle is valid. The command returns boolean 0 or 1. Optionally, the reaction may be filtered by a standard filter list, and if it does not pass the filter, it is reported as not valid. If filters in the filter list operate on ensembles, it is sufficient if a single ensemble of the reaction passes the filter.

Example:

```
reaction exists $xhandle solvent
```

Check whether the reaction with the handle in variable **`$xhandle`** exists and, if it exists, whether it contains a solvent ensemble.

reaction expand

```
reaction expand xhandle ?allowambigous? ?noimplicith?
x.expand(?allowambiguous=?,?noimplicith=?)
```

This command expands all superatoms in the ensembles of the reaction. The mechanisms for the expansion of superatoms are described in detail for the **`atom expand`** command. This command is functionally equivalent, working on all atoms in all of the reaction ensembles instead a single atom.

Example:

```
reaction expand $xhandle
```

The command returns the total number of successfully expanded atoms in all reaction ensembles.

reaction expr

```
reaction expr xhandle expression
x.expr(expression)
```

Compute a standard **SQL**-style property expression for the reaction. This is explained in detail in the chapter on property expressions.

reaction filter

```
reaction filter xhandle filterlist
x.filter(filters)
```

Check whether the reaction passes a filter list. The return value is 1 for success and 0 for failure.

Example:

```
reaction filter [reaction create {C=C>[Pt]>CC}] platinum
```

checks whether the reaction involves a platinum atom in any role.[2] If the filter operates on ensembles or minor objects, it is sufficient to have a single ensemble or ensemble minor object pass the filter condition.

reaction forget

```
reaction forget xhandle ?objclass?
x.forget(?objectclass=?)
```

This command is essentially the same as the **`ens forget`** command. It is applied to all ensembles in the reaction.

The command returns the original ensemble handle or reference.

reaction get

```
reaction get xhandle propertylist ?filterset? ?parameterdict?
reaction get xhandle attribute
x.get(property=,?filters=?,?parameters=?)
x.get(attribute)
x[property/attribute]
x.property/attribute
Reaction.Get(data,property=,?filters=?,?parameters=?)
Reaction.Get(data,attribute)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

Examples:

```
reaction get $xhandle {X_IDENT X_NAME}
```

yields the ID and name of the reaction as a list. If the information is not available, an attempt is made to compute it. If the computation fails, an error results.

```
reaction get $xhandle {E_FORMULA E_WEIGHT}
```

reports the formula and molecular weight of all reaction ensembles. The result is delivered as a nested list. The first list contains the atomic formulae, the second list contains the weights.

Currently, it is not possible to use filters with this command (and the other retrieval command variants) which are not operating directly on the reaction object, but on objects lower in the hierarchy such as ensembles or atoms.

2. A filter testing for the presence of any element is automatically created when an element name is used as a filter name. Element filters except for the most common ones (*carbon*, etc.) do not show up in the filter list before they are instantiated.

For the use of the optional property parameter list argument, refer to the documentation of the **`ens get`** command.

Instead of using a property name or a reaction attribute, it is also possible to use any valid value of property `E_REACTION_ROLE` to directly access specific ensembles of the reaction. The allowed values are dynamically read from the property definition and thus customizable.

Example:

```
set ehp [reaction get $xh product]
set ehp [reaction ens $xh product]
```

These commands yield the same results, but the first directly tests the values of `E_REACTION_ROLE` of the reaction ensembles, while the second one uses the predefined *product* filter.

Variants of the **`reaction get`** command are **`reaction new, reaction dget, reaction jget, reaction jnew, reaction jshow, reaction nget, reaction show, reaction sqldget, reaction sqlget, reaction sqlnew,`** and **`reaction sqlshow`**.

In addition to property data and reaction role selectors, a reaction object possesses a few attributes, which can be retrieved with the *get* command (but not by its related sister subcommands like **`dget`***,* **`sqlget`***,* etc.). Some of them are also modifiable via **`reaction set.`** These attributes are:

- *coords*
  If the toolkit was compiled with factory support, these are the coordinates of the object icon on its workbench, encoded as integer pair. This attribute can be changed.
- *deletable*
  Flag indicating whether the object can be deleted with a standard **`reaction delete`** command. This attribute is read-only. Objects which are, for example, property data values or a part of a **`molfile loop`** command cannot be deleted by standard means.
- *failures*
  If the property computation failure cache is active, return a list of all properties which have failed computation for this object after the last structural change. This attribute is read-only.
- *footer*
  If the toolkit was compiled with factory support, this is the footer of the object icon on a workbench. This attribute can be changed.
- *gflags*
  If the toolkit was compiled with factory support, this is the currently set object icon rendering flag collection.
- *header*
  If the toolkit was compiled with factory support, this is the header of the object icon on a workbench. This attribute can be changed.
- *hidden*
  Flag indicating whether the object is hidden. This is not the same as the *invisible* state. This attribute is intended to be used for rendering selections. This attribute can be changed.
- *incomplete*
  Boolean status flag indicating an aborted input operation during the read of the object from file, which returned the structure intact but without the complete set of associated data. An aborted input may be either be the result of an explicitly set input control flag, or by encountering property data which could not be decoded. This attribute is read-only.

- *invisible*
  Flag indicating whether the object is invisible. This is not the same as the *hidden* state. An invisible object is no longer accessible via its handle. This is usually the case for objects which are scheduled for deletion, but still have lingering pointer references. This attribute is read-only.
- *javaobject*
  If the toolkit was compiled with **JNI** support, this attribute reports the memory address of the **JNI** wrapper class instance, if it exists.
- *loopitem*
  A read-only attribute returning the handle of the currently active iterator item.
- *modcount*
  Object data modification count. This attribute is read-only.
- *mutexcount*
  The number of recursive mutex locks held for this object. Only supported on Linux.
- *pyobject*
  If the toolkit was compiled with Python support, this attribute reports the memory address of the Python wrapper class instance, if it exists. This attribute is read-only.
- *pyrefcount*
  If the toolkit was compiled with Python support, this attribute reports the reference count of the Python wrapper class instance, if it exists. This attribute is read-only.
- *record*
  The current iterator record (starting with 1) of the reaction. It is possible to set the value and thus skip or revisit reaction ensembles in the iterator.
- *refcount*
  If the **Tcl** interpreter is using native **Cactvs** objects instead of string-based major object handles and integer-based minor object labels to identify toolkit objects, this returns the number of **Tcl** object references active for this ensemble. This attribute is read-only.
- *scoped*
  A boolean object visibility control flag. If set, and global control flag `::cactvs(object_scope)` is also set, the object is visible only in the**Tcl** interpreter which set the scope flag and thus claimed it. Object list commands executed in other interpreters omit this object, and attempts to decode its handle in other interpreters will fail. The most common use of this feature is the hiding of persistent chemistry objects in scripted property computation functions.
- *selected*
  Flag indicating whether the object is selected. This attribute can be changed.
- *tooltip*
  If the toolkit was compiled with factory support, this is the tooltip of the object icon on a workbench. This attribute can be changed.
- *uuid*
  An automatically generated **UUID** globally identifying the object. This attribute is read-only, different for every object, and not dependent on its contents.
- *x*
  If the toolkit was compiled with factory support, this is the *x* coordinate of the object icon on its workbench. This attribute can be changed.

- *y*
  If the toolkit was compiled with factory support, this is the *y* coordinate of the object icon on its workbench.This attribute can be changed.

reaction getparam

```
reaction getparam xhandle property ?key? ?default?
x.getparam(property=,?key=?,?default=?)
```

Retrieve a named computation parameter from valid property data. If the key is not present in the parameter list, an empty string is returned (`None` for **Python**). If the default argument is supplied, that value is returned in case the key is not found.

If the key parameter is omitted, a complete set of the parameters used for computation of the property value is returned in dictionary format.

This command does not attempt to compute property data. If the specified property is not present, an error results.

Example:

```
reaction getparam $xhandle X_GIF format
```

returns the actual format of the image, which could be **GIF**, **PNG**, or various bitmap formats.

reaction hadd

```
reaction hadd xhandle ?filterset? ?flags? ?changeset?
x.hadd(?filters=?,?flags=?,?changeset=?)
```

Add a standard set of hydrogens to the ensembles of the reaction. If the *filterset* parameter is specified, only those atoms which pass the filter set are processed.

Additional operation flags may be activated by setting the *flags* parameter to a list of flag names, or a numerical value representing the bit-ored values of the selected flags. By default, the flag set is empty, corresponding to the use of an empty string or *none* as parameter value. These flags are currently supported:

- *keepflags*
  For expert use only. Do not discard min/max values and property scope flags for atom properties when hydrogen is added.
- *no2dcoords*
  Do not assign 2D coordinates to the added hydrogens, even if the rest of the atoms in the ensemble have valid 2D coordinates. In any case, 2D coordinates are never added when the reaction ensemble does no already possess valid 2D coordinates.
- *no3dcoords*
  Do not assign 3D coordinates to the added hydrogens, even if the rest of the atoms in the ensemble have valid 3D coordinates. In any case, 3D coordinates are never added when the reaction ensemble does no already possess valid 3D coordinates.
- *noanions*
  Do not add hydrogen to atoms with a negative formal charge.
- *noatoms*
  Do not add hydrogen to atoms without any bonds.

- *nocations*
  Do not add hydrogen to atoms with a positive formal charge.
- *noelements*
  Do not add hydrogen if the reaction ensemble consists purely of isolated metal atoms, which probably represent the material in elementary form, or as an alloy.
- *noexcessvalences*
  Similar to *nohighvalences*, but hydrogen is not added to any atom which is not in its lowest standard bonded valence state.
- *nofixatomtext*
  Do not adjust property `A_TEXTLABEL` (if present) by removing references to implicit H from it on atoms where hydrogen is added. For example, by default "NHCOOEt" becomes "NCOOEt" after adding an instantiated hydrogen to the nitrogen atom. This reduces confusion on the hydrogen status when rendering all atoms.
- *nohighvalences*
  Do not add hydrogen to atoms which already exceed their lowest standard valence minus any formal charge. This option only applies to elements which have a defined lowest standard valence (this is configurable via the element table).
- *nomemory*
  Do not remember the added hydrogen atoms as automatically added. Normally, a flag is retained as part of the atom information which distinguishes atoms which were added by automatic processing, such as hydrogen addition, from those which were originally input.
- *nometals*
  Do not attempt to add hydrogen to atoms which are metals (as defined in the system element table).
- *nospecial*
  Do not perform hydrogen addition to atoms which participate in non-standard bonds (all bonds with `B_TYPE` not *normal*).
- *protonate*
  Add a single proton to the first suitable atom. The charge of the atom is increased and only a single hydrogen is added regardless of the standard number of missing hydrogens, and this command *does* issue the standard property invalidation event for atom and bond changes. In the reaction command variant, this option is rarely useful. It is supported for compatibility with the **`atom hadd`** command.
- *resetmemory*
  Reset the origin flag described above for all atoms in the reaction ensembles. All current atoms appear to be part of the original atom set.

Adding hydrogens with this command is less destructive to the property data set of the reaction ensembles than adding them with individual **`atom create/bond create`** commands, except in case the *protonate* flag is set, because many properties are designed to be indifferent to explicit hydrogen status changes, but are invalidated if the structure is changed in other ways.

If the effects of the hydrogen addition step to the validity of the property data set should not be handled according to this standard procedure, it is possible to explicitly generate additional property invalidation events by specifying an event list as the optional last parameter, for example a list of *atom* and *bond* to trigger both the atom change and bond change events.

The command returns the number of hydrogens which were added to all reaction ensembles.

Example:

```
set xhandle [reaction create {[C]=[C]>>[C]-[C]}]
reaction hadd $xhandle
```

adds a total of ten hydrogens to the two reaction ensembles, transforming them into hydrogen-complete ethene and ethane.

reaction hdup

```
reaction hdup xhandle ?dataset? ?position?
x.hdup(?target=?,?position=?)
```

This command performs the same operation as the **`reaction dup`** command, but additionally adds a standard set of hydrogens to all ensembles of the duplicated reaction.

reaction hierarchy

```
reaction hierarchy xhandle ?filterlist? ?root?
x.hierarchy(?filters=?,?root=?)
```

Return the hierarchy handle or reference of the hierarchy the reaction is part of. If the reaction is not member of a hierarchy, or does not pass all of the optional filters, an empty string or **`None`** for **PYTHON** is returned. By default, the hierarchy object which directly contains the reaction is returned. If the *root* flag is set, the root hierarchy object is reported instead, which is the same only if the hierarchy has only a single level.

Example:

```
reaction hierarchy $xhandle
```

reaction hstrip

```
reaction hstrip xhandle ?flags? ?changeset?
x.hstrip(?flags=?,?changeset=?)
```

This command removes hydrogens from all ensembles in the reactions. By default, all hydrogen atoms on the ensemble are removed.

The *flags* parameter can be used to make the operation more selective. It may be a list of the following flags:

- *deprotonate*
  If this flag is set, a single proton is removed from the first suitable atom. This command variant triggers a standard atom and bond change property invalidation event, and it always ends processing after removing the first proton. Proton removal decreases the charge of the atom by one. In the reaction command variant, this flag is rarely useful - it is supported for compatibility with the **`atom hstrip`** command
- *keepalphawedge*
  Keep hydrogen atoms which are bonded to an atom which is at the tip of a wedgebond. This flag excludes the case where the bond to the hydrogen atom is the wedge bond - use the *keepwedge* flag to cover this case.
- *keepisotopes*
  Keep hydrogen atoms which are isotope labels (including enriched/depleted ^{1}H).

- *keeporiginal*
  Hydrogen atoms which were not automatically added via a *hadd* command are retained. Note that hydrogen addition commands can be run in a mode which does not leave information about automatic addition - hydrogens added this way will also survive.
- *keepprotons*
  Keep any molecules which consist only of hydrogen atoms (such as protons, hydride anions, and molecular hydrogen).
- *keepspecial*
  If this flag is set, hydrogens which are usually displayed, such as on aldehydes, wedge bonds, carbon triple bonds or hetero atoms are retained.
- *keepwedge*
  keep hydrogens which are at the end of a wedge bond, indicating stereochemistry.
- *normalize*
  Normalize the wedge pattern for standard cases, removing wedges from hydrogens if the result is still stereochemically defined. Hydrogens which lose their wedge in this process are no longer protected by the *keepwedge* flag.
- *wedgetransfer*
  If a hydrogen atom is removed which is at the end of a wedge, the wedge information is saved by transferring the wedge (changing its up/down status if necessary) to an adjacent, surviving bond. This flag has no effects if the *keepspecial* or *keepwedge* flags are set. This flag is set by default.

If the *flags* parameter is an empty string, or *none*, it is ignored. The default flag value is *wedgetransfer* - but the default value is overridden if any flags are set!

If the *changeset* parameter is used, all property change events listed in the parameter are triggered.

Hydrogen stripping is not as disruptive to the data content of the reaction ensembles as normal atom deletion, except in case the *deprotonate* flag is set. The system assumes that this operation is done as part of some file output or visualization preparation. However, if any new data is computed after stripping, the computation functions see the stripped structure, and proceed to work on that reduced structure without knowledge that there are implicit hydrogens.

The command returns the total number of hydrogens stripped from all reaction ensembles.

Example:

```
reaction hstrip $xhandle [list keeporiginal wedgetransfer]
```

reaction hydrogenate

```
reaction hydrogenate xhandle ?filterset? ?changeset?
x.hydrogenate(?filters=?,?changeset=?)
```

Reduce all bonds in the reaction to single bonds, except those excluded by the filter set.

If a change set is supplied, its interpretation is the same as in **`reaction hadd`**.

The command returns the number of added hydrogens.

reaction index

```
reaction index xhandle
```

```
x.index()
```

Get the position of the reaction in the object list of its dataset. If the reaction is not member of a dataset, -1 is returned.

reaction jget

```
reaction jget xhandle propertylist ?filterset? ?parameterdict?
x.jget(property=,?filters=?,?parameters=?)
Reaction.Jget(data,property=,?filters=?,?parameters=?)
```

This is a variant of **`reaction get`** which returns the result data as a **JSON** formatted string instead of **TCL** or **PYTHON** interpreter objects. The command is usable only for property data, not attribute retrieval.

reaction jnew

```
reaction jnew xhandle propertylist ?filterset? ?parameterdict?
x.jnew(property=,?filters=?,?parameters=?)
Reaction.Jnew(data,property=,?filters=?,?parameters=?)
```

This is a variant of **`reaction new`** which returns the result data as a **JSON** formatted string instead of **TCL** or **PYTHON** interpreter objects.

reaction jshow

```
reaction jshow xhandle propertylist ?filterset? ?parameterdict?
x.jshow(property=,?filters=?,?parameters=?)
Reaction.Jshow(data,property=,?filters=?,?parameters=?)
```

This is a variant of **`reaction show`** which returns the result data as a **JSON** formatted string instead of **TCL** or **PYTHON** interpreter objects.

reaction ldup

```
reaction ldup ?xhandlelist?...
Reaction.Ldup(?exref/xrefsequence?,...)
```

Duplicate all reactions in the argument list(s) in default mode.

The return value is a single list (even if multiple source lists are used) of the duplicated reaction handles or references. If an argument list element is an empty string (or **`None`** for **PYTHON**), it indicates a missing object, and the output list also receives an empty string element (for **TCL**) or **`None`** (for **PYTHON**) at its position, without raising an error.

reaction lhdup

```
reaction lhdup ?xhandlelist?...
Reaction.Lhdup(?xref/xrefsequence?,...)
```

Duplicate all reaction in the argument list(s) in default mode, and add hydrogens.

The return value is a single list (even if multiple source lists are used) of the duplicated reaction handles or references. If an argument list element is an empty string (or **`None`** for **PYTHON**), it indicates a missing object, and the output list also receives an empty string element (for **TCL**) or **`None`** (for **PYTHON**) at its position, without raising an error.

reaction link

```
reaction link xhandle ?xhandle?...
x.link(?xhandle/xref?,...)
```

Link the reaction to any number of additional reactions, creating a reaction network in the process. Duplicate links, or links to reactions which already exist are ignored. A reaction cannot link to itself.

The command returns the number of newly created reaction links.

reaction links

```
reaction links xhandle ?filterlist? ?recursive?
x.links(?filters=?,?recursive=?)
```

Return the set of reactions this reaction is linked to as a list of reaction handles or reaction references. The set may optionally be filtered by standard reaction-level filter conditions. If the *recursive* boolean flag is set linked reactions beyond the first neighbor sphere are also included, with duplicates of reactions in closer spheres removed.

reaction list

```
reaction list ?filterlist?
Reaction.List(?filters=?)
```

This command returns a list of the reaction handles currently registered in the application. This list may optionally be filtered by a standard filter list. If the filter operates on the reaction ensembles and not on the reaction object, it is sufficient if a single reaction ensemble passes the filter.

Example:

```
reaction list solvent
```

lists the handles of all reactions in the application which contain a solvent ensemble.

reaction lock

```
reaction lock xhandle propertylist/reaction/all ?compute?
x.lock(property=,?compute=?)
```

Lock property data of the reaction, meaning that it is no longer subject to the standard data consistency manager control. The data consistency manager deletes specific property data if anything is done to the reaction which would invalidate the information. Blocking the consistency manager can be useful when building reactions from components in a script. Property data remains locked until is it explicitly unlocked.

The property data to lock can be selected by providing a list of the following identifiers:

- Property names
  Valid property instances on the reaction, reaction ensembles, or ensemble minor objects are locked. If the boolean *compute* flag is set, an attempt is made to compute the property if it is not yet present. Otherwise, a request to lock non-existent data is silently ignored. It is not possible to lock individual property fields.
- *all*
  All valid reaction properties are locked. Ensemble properties and ensemble minor object properties are not affected. The compute flag is ignored.

- *reaction*
  This is an object class identifier. All property data which is controlled by the reaction major object and attached to the specified object class is locked. Since reactions do not contain minor objects, this identifier is equivalent to *all*.

The lock can be released by a **`reaction unlock`** command.

The return value is the original reaction handle or reference.

Example:

```
set xhandle [reaction create {C=C>[Pt]>CC}]
reaction lock $xhandle X_GIF 1
reaction clear $xhandle agent
reaction unlock $xhandle X_GIF
```

In this example, first a reaction depiction in property data `X_GIF` is generated and locked. After that, the reaction and reaction ensembles can be manipulated without losing the image data. The agent is removed in the next step - but the `X_GIF` image which shows the catalyst and which normally would have been deleted when the agent is removed is kept. Finally, the image property data is put back under the standard control of the data consistency manager.

reaction loop

```
reaction loop xhandle objvar ?maxrec? ?offset? body
x.loop(function=,?maxloop=?,?offset=?,?variable=?)
for e in x:
```

Loop over the ensembles in a reaction. This command is similar to **`molfile loop`**. On each iteration, the variable is set to the handle of the current member object, and then the body code is executed. The loop object is an original ensemble, not a duplicate.

Ensembles should not be removed from the reaction during the loop.

If a maximum record count is set, the loop terminates after the specified number of iterations. If the maximum record argument is set to an empty string, a negative value, or *all*, the loop covers all reaction elements. This is also the default. The return value is the number of calls to the body code block.

For **Tcl** scripts, within the loop, the standard **Tcl** **`break`** and **`continue`** commands work as expected. If the body script generates an error, the loop is exited.

If no offset is specified, the loop starts at the first element. Within the loop body, the reaction attribute *record* is continuously updated to indicate the current loop position. Its value starts with one, like file records in the **`molfile loop`** command.

The **Python** version of the loop method does intentionally have a different argument sequence for convenience. The function argument may either be a multi-line string (similar to the **Tcl** construct), or a function reference. Functions are called with the reference of the current loop ensemble as single argument, and have their own context frame, so that the specification of a reference variable is not generally useful in that call style, though is is allowed. For string function blocks the code is executed in the local call frame, and the variable with the current object reference is visible locally. Script code blocks must be written with an initial indentation level of zero. Within the **Python** functions, the normal *break* and *continue* loop control commands cannot be used to to scope limitations. Instead, the custom exceptions *BreakLoop* and *ContinueLoop* can be raised. These are automatically caught and processed in the loop body handler code.

In **Python**, there is also an object iterator so that simple loops over reaction ensemble can be written with a `for` statement. The reaction object iterator is of the *self* style (i.e. there is one per reaction, these are not independent objects), so nesting them is not possible on the same reaction.

Python object loop constructs and their peculiarities are discussed in more detail in the general chapter on **Python** scripting.

Example:

```
reaction loop $xh eh {
   puts „[ens get $eh E_REACTION_ROLE] at position[reaction get $xh record]‟
}
```

reaction max

```
reaction max xhandle propertylist ?filterset?
x.max(property=,?filters=?)
```

Get the maximum values of the properties named in the *propertylist* parameter. The return value of the command is a list of the maximum property values. While it is possible to work with reaction properties, this is pointless since there is only a single instance of a reaction property per reaction. Usually, ensemble or ensemble minor object properties are retrieved. The objects whose property values are used for the determination of the maximum values may optionally be filtered by a standard filter set.

Example:

```
reaction max $xhandle E_WEIGHT {1 reagent product}
```

computes the maximum molecular weight from the reagent and product ensembles, ignoring other reaction ensembles such as solvents or catalysts.

reaction merge

```
reaction merge xhandle ?xhandle?...
x.merge(?xhandle?,...)
```

Merge individual reactions into a multi-step reaction with intermediates. The current product ensemble of the target reaction object (first reaction handle in `Tcl`) gets reassigned as intermediate in `E_REACTION_ROLE`. Next, for every merged reaction, its intermediates and then the product are duplicated and appended to the target reaction. The merged source reactions remain unchanged. No check is performed that the original target reaction product ensemble and the reagent ensemble of a merged reaction are compatible.

If the merge reactions are specified, the command does nothing. It is not possible to merge a reaction with itself.

The command result is the target ensemble handle or reference.

Example:

```
set xh1 [reaction create CCO>>C=C]
set xh2 [reaction create C=C.ClCl>>ClCCCl]
reaction merge $xh1 $xh2
```

The result is a reaction with ethanol as reagent, ethene as intermediate, and dichloroethane as product. The explicit chlorine reagent of the second reaction is not represented in the result.

reaction metadata

```
reaction metadata xhandle property ?field ?value??
x.metadata(property=,?field=?,?value=?)
```

Obtain property metadata information, or set it. The handling of property metadata is explained in more detail in its own introductory section. The related commands **`reaction setparam`** and **`reaction getparam`** can be used for convenient manipulation of specific keys in the computation parameter field. Metadata can only be read from or set on valid property data.

Valid field names are *bounds*, *comment*, *info*, *flags*, *parameters* and *unit*.

Examples:

```
array set gifparams [reaction metadata $xhandle X_GIF parameters]
reaction metadata $xhandle X_CONDITIONS comment "Check with legal dept whether
dehydration in molten Pu metal requires special regulatory approval"
```

The first line retrieves the computation parameters of the property `X_GIF` as keyword/value pairs. These are read into the array variable **`gifparams`**, and may subsequently be accessed as **`$gifparams(format)`**, **`$gifparams(height)`**, etc. The second example shows how to attach a comment to a property value.

reaction min

```
reaction min xhandle propertylist ?filterset?
x.min(propery=,?filters=?)
```

Get the minimum values of the properties named in the *propertylist* parameter. The return value of the command is a list of the minimum property values. While it is possible to work with reaction properties, this is pointless since there is only a single instance of a reaction property per reaction. Usually, ensemble or ensemble minor object properties are retrieved. The objects whose property values are used for the determination of the minimum values may optionally be filtered by a standard filter set.

Example:

```
reaction min $xhandle E_WEIGHT {1 reagent product}
```

computes the minimum molecular weight from the reagent and product ensembles, ignoring other reaction ensembles such as solvents or catalysts.

reaction move

```
reaction move xhandle ?datasethandle|remotehandle? ?position?
x.move(?target=?,?position=?)
```

Make the reaction a member of a dataset, or remove it from a dataset. If the dataset handle or reference parameter is omitted, or is an empty string, or **`None`** for **PYTHON**, the object is removed from its current dataset. The dataset handle or reference may be the name of a remote dataset for moving objects over a network connection.

If a target dataset handle or reference is specified, the object is added to the dataset, if allowed by the acceptance bits of the dataset, and removed from any dataset it was member of before the execution of the command. By default the object is added to the end of the dataset object list, but the final optional parameter allows the specification of a dataset object list index. The first position is index zero. If the parameter value *end* is used, or the index is bigger than the current number of dataset objects minus one, the object is appended as per the default. It is legal to use this command for moving objects within the same dataset.

Another special position value is *random* or *rnd*. This value moves to the object to a random position in the dataset. Using this mode with remote datasets is currently not supported.

The dataset handle cannot be a transient dataset.

The return value of the command is the dataset of the object prior to the move operation. It is either a dataset handle/reference, or an empty string (**Tcl**) or `None` (**Python**) if it was not member of a dataset.

Examples:

```
reaction move $xhandle $dhandle 0
reaction move $xhandle
```

In the first sample line, the reaction is inserted as the first element in a dataset. The second line reverts this operation and removes the reaction from the dataset.

This command interacts with the insert control mechanism of size-constrained datasets. More information is provided in the description of the *sizecontrol* dataset parameter.

This command can be used with a remote dataset descriptor. In that case, the reaction is packed into a serialized object representation, transmitted over the network and restored as member of the remote dataset at the specified position. The local reaction is deleted if the transfer succeeds.

Example:

```
reaction move $xhandle blockbuster@server2:9998 end
```

This command moves the reaction to the dataset which was set up as listener on port 9998 and pass phrase *blockbuster* on host *server2*. The local reaction is deleted, and its copy is inserted at the end of the remote dataset.

reaction mutex

```
reaction mutex xhandle mode
x.mutex(mode)
```

Manipulate the object mutex. During the execution of a script command, the mutex of the major object(s) associated with the command are automatically locked and unlocked, so that the operation of the command is thread-safe. This applies to builds that support multi-threading, either by allowing multiple parallel script interpreters in separate threads or by supporting helper threads for the acceleration of command execution or background information processing.

This command locks major objects for a period of time that exceeds a single command. A lock on the object can only be released from the same interpreter thread that set the lock. Any other threaded interpreters, or auxiliary threads, block until a mutex release command has been executed when accessing a locked command object. This command supports the following modes:

- *lock*
  Increase the recursive mutex lock count on the object. The command returns the current lock count after the command, excluding the transient single-command lock.

- *reset*
  Release all persistent locks on the object, if they exist.
- *test*
  Return the current persistent lock count on the object. This excludes the transient per-command lock.
- *unlock*
  Decrease the recursive lock count on the object. The command returns the current lock count after the command, excluding the transient single-command lock. Unlocking an object which has not been persistently locked results in an error.

There is no *trylock* command variant because the command already needs to be able to acquire a transient object mutex lock for its execution.

The command returns the current lock count.

reaction need

```
reaction need xhandle propertylist ?mode? ?parameterdict?
x.need(property=,?mode=?,?parameters=?)
```

Standard command for the computation of property data, without immediate retrieval of results. This command is explained in more detail in the section about retrieving property data.

The return value is the original reaction handle or reference.

Example:

```
reaction need $xhandle E_WEIGHT recalc
```

reaction new

```
reaction new xhandle propertylist ?filterset? ?parameterdict?
x.new(property=,?mode=?,?parameters=?)
Reaction.New(data,property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **`reaction get`** command. The difference between **`reaction get`** and **`reaction new`** is that the latter forces the re-computation of the property data, regardless whether it is present and valid, or not.

reaction nget

```
reaction nget xhandle propertylist ?filterset? ?parameterdict?
x.nget(property=,?mode=?,?parameters=?)
Reaction.Nget(data,property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **`reaction get`** command. The difference between **`reaction get`** and **`reaction nget`** is that the latter always returns numeric data, even if symbolic names for the values are available.

reaction nnew

```
reaction nnew xhandle propertylist ?filterset? ?parameterdict?
x.nnew(property=,?filters=?,?parameters=?)
Reaction.Nnew(data,property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading object data and attributes. It is explained in more detail in the section about retrieving property data.

For examples, see the **`reaction get`** command. The difference between **`reaction get`** and **`reaction nnew`** is that the latter always returns numeric data, even if symbolic names for the values are available, and that property data re-computation is enforced.

reaction nitrostyle

```
reaction nitrostyle xhandle style
x.nitrostyle(style=)
```

Change the internal encoding of nitro groups and similar functional groups in the ensembles of the reaction. Possible values for the style parameter are:

- *asis* No change
- *ionic* Change to encoding to a positive charge on the center atom, and a negative on one of the oxygens
- *xionic* As above, but also change the encoding of azides, etc.
- *neutral* Change the encoding to the neutral form with extended valence. *pentavalent* is an alias.
- *xneutral* As above, but also change the encoding of azides, etc.

The command returns the original reaction handle or reference.

reaction pack

```
reaction pack xhandle ?maxsize? ?requestprops? ?suppressedprops? ?compressionlib?
x.pack(?maxsize=?,?requestprops=?,?suppressedprops=?,?compressionlib=?)
```

Pack the reaction object into a base64-encoded compressed serialized object string. This string does not contain any non-printable characters and is a full dump of the internal state of the object, omitting only property data that was declared to be so easily re-computed that a dump is not worthwhile. The reaction ensembles and their property data are part of the dump. Further object relationships, such as datasets the reaction or reaction ensembles might be a member in, or tables the reaction or its ensembles are associated with are not saved.

The maximum size of the object string (default -1, meaning unlimited size) can be configured by the optional *maxsize* parameter. The size is specified in bytes. If the pack string would be longer than the maximum size, an error results.

The other optional property parameter lists allow to request a specific property set to be part of the package, even if it normally would not be included, and to explicitly omit properties from the dump. No property computation is performed, and suppressed properties are not purged from the reaction.

Reactions can be restored from a packed object string by the **`reaction unpack`** or **`reaction create`** commands.

The reaction object and its ensembles remain in existence after using this command.

The default compression library is *zlib*. Other useful variants include *lzo* and *gzip* (and there are other internal types), but these may not be available on all builds due to license issues, and you need to specify the compression library when a dataset is unpacked. It is generally recommended to stay with *zlib*.

The return value of this command is the packed string.

In **Python**, reactions support the standard *pickle/unpickle* protocol.

Example:

```
set dbstring [reaction pack [reaction create CC=O>>CCO]]
```

reaction properties

```
reaction properties xhandle ?pattern? ?intersectionmode?
x.properties(?pattern=?,?intersectionmode=?)
```

Get a list of valid properties of the reaction proper and the reaction ensembles. By default, reaction properties (prefix `X_`), dataset properties (prefix `D_`), as well as the properties of the ensembles in the reaction (prefix `E_`) and the properties of their minor objects (atoms, bonds, etc.) are listed.

Property subsets may be selected by a non-empty filter pattern. In case of reaction ensemble or minor ensemble object properties which are not present in all reaction ensembles, the default intersect mode is *union*, meaning that all properties are reported for which at least one instance exists. The alternative mode *intersect* only lists those ensemble properties which are present in all reaction ensembles.

This command may also be invoked as **`reaction props`** or **`x.props()`**.

Example:

```
reaction properties $xhandle X_*
reaction props $xhandle E_* intersect
```

The first example returns a list of the currently valid reaction properties. The second example lists all reaction properties which are present in all reaction ensembles.

reaction purge

```
reaction purge xhandle propertylist/reaction/specialname ?emptyonly?
x.purge(properties=,?emptyonly?)
```

Delete property data from the reaction. The properties may be reaction properties (prefix `X_`), dataset properties (prefix `D_`) or properties of the reaction ensembles, such as ensemble or atom properties. If a property marked for deletion is not present on an object, it is silently ignored. If the reaction is not a dataset member, a request for the deletion of dataset properties is also ignored.

If the object class name *reaction* is used instead of a specific property name, all reaction property data (`X_` prefix) is deleted from the reaction.

If another object class name, such as *ens* or *atom*, is used instead of a property name, all properties of that class set on the reaction ensembles are deleted, if they are not locked, or filtered out by the optional empty-only flag.

Besides normal property and class names, a few convenient special names for common property deletion tasks on the ensembles of the reaction are defined and can be used as a replacement for the property list. These include:

- *atomquery*
  Delete all atom query information from the reaction ensembles (`A_QUERY` and other query properties).
- *atomstereochemistry*
  Delete all atomic atom stereo descriptors from the reaction ensembles, but keep those for bonds.
- *bondquery*
  Delete all bond query information from the reaction ensembles (`B_QUERY` and other query properties).
- *bondstereochemistry*
  Delete all bond stereo descriptors from the reaction ensembles, but keep those for atoms.
- *isotopes*
  Delete isotope information in `A_ISOTOPE` and other isotope properties which may be defined in future software versions.
- *query*
  Delete all query information from the reaction ensembles (`A_QUERY`, `B_QUERY` and other query properties).
- *radicals*
  Delete atomic radical information in `A_RADICAL` and other radical-related properties which may be defined in future software versions.
- *stereochemistry*
  Delete all stereochemistry descriptors, including 2D wedges, but not 3D coordinates. The implicit property list includes `A_LABEL _STEREO, B_LABEL_STEREO, A_CIP_STEREO, B_CIP_STEREO, A_DL_STEREO, B_CISTRANS_STEREO, A_HASH_STEREO, B_HASH_STEREO, A_MAP_STEREO`, `B_MAP_STEREO`, `A_STEREOINFO`, `B_STEREOINFO`, `A_STEREO_GROUP`, `M_STEREO_COUNT`, `E_STEREO_COUNT` and `B_FLAGS` (only selected bits, the property remains valid if present).
- *wedges*
  Delete wedge bond flags in property `B_FLAGS`. If `B_FLAGS` is not present, the command is ignored and no computation attempt is made.

The optional boolean flag *emptyonly* can be used to restrict the deletion to those properties where all the values for a property associated with a major object (such as on all atoms in an ensemble for atom properties, or just the single ensemble property value for ensemble properties) are set to the default property value.

The return value is the original reaction handle or reference.

Examples:

```
reaction purge $xhandle X_IDENT
reaction purge $xhandle E_IDENT 1
```

The first example deletes the property data `X_IDENT` for the selected reaction if it is present. The second example deletes property `E_IDENT` from all ensembles in the reaction if the property value on that ensemble is equal to the default value for `E_IDENT`.

reaction ref

```
Reaction.Ref(identifier)
```

Python only method to get a reaction reference from a handle or another identifier. For reactions, other recognized identifiers are reaction references, or integers encoding the numeric part of the handle string.

reaction remove

```
reaction remove xhandle ?enslist?...
x.remove(?erefsequence/eref?,...)
```

Remove the ensembles in the ensemble lists from the reaction. If an ensemble from the list is not part of the reaction, it is ignored. Removed ensembles are not destroyed and remain accessible via their handles.

The command returns the number of removed ensembles.

Examples:

```
reaction remove $xhandle $ehandle
reaction remove $xhandle [reaction ens $xhandle]
```

The first example removes the ensemble from the reaction if it is part of the reaction. The second example removes all ensembles from the reaction - this is essentially the same as **`reaction clear`**.

reaction rename

```
reaction rename xhandle srcproperty dstproperty
x.rename(srcproperty=,dstproperty=)
```

This is a variant of the **`reaction assign`** command. Please refer the command description in that paragraph.

reaction reorder

```
reaction reorder xhandle
x.reorder()
```

Sort the ensembles in the reaction into standard sequence (*reagent/product/solvent/catalyst/intermediate/impurity/byproduct/agent/waste*) as defined in property `E_REACTION_ROLE`. In addition, empty *reagent* and *product* (but not *solvent*, etc.) ensembles are automatically created for the reaction in case they are not present.

The command returns the original ensemble handle. or reference.

reaction reverse

```
reaction reverse xhandle
x.reverse()
```

Reverse the direction of the reaction. Only the roles of ensembles with *reagent* and *product* roles are changed.

The command returns the original reaction handle.

reaction scan

```
reaction scan xhandle expression/queryhandle ?mode? ?parameters?
x.scan(query=,?mode=?,?parameters=?)
```

Perform a query on the reaction object. The syntax of the query expression and the optional selection list is the same as that of the **dataset scan** command with a transient dataset consisting of the current reaction only. For more details, please refer to the paragraphs on **dataset scan, ens scan** and **molfile scan**.

The return value depends on the mode. The default query mode, different from the default in **dataset scan**, is *exists*.

In case the query contains ensemble structure match conditions which are not part of a reaction query, or there are ensemble data retrieval specifications, these are tested on and applied to the *reagent* ensemble of the reaction. Reactions which do not possess a reagent ensemble (half reactions, etc.) are ignored for these types of test.

reaction set

```
reaction set xhandle ?property value?...
x.set(property,value,...)
x.set({property:value,...})
x.property = value
x[property] = value
```

Standard data manipulation command. It is explained in more detail in the section about setting property data.

Examples:

```
reaction set $xhandle X_NAME "New multi-component reaction"
reaction set $xhandle E_IDENT "X-124"
```

The first line is a simple set operation for a reaction property. The second line shows how to set properties of multiple ensembles in one step. The same property value is assigned to all ensembles.

reaction setparam

```
reaction setparam xhandle property ?key value?...
reaction setparam xhandle property dictionary
x.setparam(property,?key,value?...)
x.setparam(property,dict)
```

Set or update a property computation parameter in the metadata parameter list of a valid property. This command is described in the section about retrieving property data. The current settings of the computation parameters in the property definition are not changed.

The return value is the updated property computation parameter dictionary.

Example:

```
reaction setparam $xhandle X_GIF comment "Top Secret"
```

reaction show

```
reaction show xhandle propertylist ?filterset? ?parameterdict?
x.show(property=,?filters=?,?parameters=?)
Reaction.Show(data,property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **reaction get** command. The difference between **reaction get** and **reaction show** is that the latter does not attempt computation of property data, but raises an error if the data is not present and valid. For data already present, **reaction get** and **reaction show** are equivalent.

The **Python** class method is a one-shot command. The transient ensemble created from the initialization items is automatically deleted when the command finishes. The data for the creation of the temporary ensemble is equivalent to the first argument of the standard constructor. Additional constructor parameters cannot be used.

reaction sort

```
reaction sort xhandle ?sort_property? ?relabel? ?duplicate? ?datasethandle?
   ?position?
x.sort(?property=?,?relabel=?,?duplicate=?,?target=?,?position=?)
```

This command applies the **ens sort** command to all reaction ensembles. Please refer to the descriptors of the **ens sort** command for an explanation of the parameters.

The command returns the original reaction handle or reference.

reaction split

```
reaction split xhandle ?minsize? ?splitproperty?
x.split(?minsize=?,?splitproperty=?)
```

Split the molecules of the ensembles in the reaction into individual ensembles. The return value is a list of all ensemble handles or references in the reaction after the operation, similar to the output of a **reaction ens** command. The initial reaction ensembles are modified, and their old handles may be reused as one of the new single-molecule ensemble handles. If an input ensemble contains only a single molecule, and that molecule passes the optional size filter, the command is a no-op for that ensemble. All result ensembles remain members of the reaction in their old reaction role.

The optional *minsize* parameter is a minimum for the number of heavy atoms (property `M_HEAVY_ATOM_COUNT`) in any of the molecules. If this is not an empty string, molecules which have less atoms than the minimum are not duplicated. If all molecules in a reaction ensemble are smaller than the required size, the ensemble is destroyed.

The optional split property argument can be used to split an ensemble on values of a molecule property, which needs to be either already set or computable, instead of simply separating fragments on connectivity. All molecules in an input ensemble which have a common value of this property are put into a joint result ensemble, and each distinct property value starts a new result ensemble. Molecules with a common property value do not need to be present in the input ensemble in a consecutive sequence, nor are there any special requirements for the data type or value range of the split property, as long as the data type has a comparison function. If the values of the split property are distinct over all molecules in an input ensemble, the outcome of command is indistinguishable from running it without any split property.

Comparison of property values is performed separately within every reaction ensemble, not across the complete ensemble set in the reaction.

reaction sqldget

```
reaction sqldget xhandle propertylist ?filterset? ?parameterdict?
x.sqldget(property=,?mode=?,?parameters=?)
Reaction.Sqldget(data,property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **`reaction get`** command. The differences between **`reaction get`** and **`reaction sqldget`** are that the latter does not attempt computation of property data, but initializes the property value to the default and returns that default, if the data is not present and valid; and that the **SQL** command variant formats the data as **SQL** values rather than for **TCL** or **PYTHON** script processing.

reaction sqlget

```
reaction sqlget xhandle propertylist ?filterset? ?parameterdict?
x.sqldget(property=,?mode=?,?parameters=?)
Reaction.Sqlget(data,property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **`reaction get`** command. The difference between **`reaction get`** and **`reaction sqlget`** is that the **SQL** command variant formats the data as **SQL** values rather than for **TCL** or **PYTHON** script processing.

reaction sqlnew

```
reaction sqlnew xhandle propertylist ?filterset? ?parameterdict?
x.sqlnew(property=,?mode=?,?parameters=?)
Reaction.Sqlnew(data,property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **`reaction get`** command. The differences between **`reaction get`** and **`reaction sqlnew`** are that the latter forces re-computation of the property data, and that the **SQL** command variant formats the data as **SQL** values rather than for **TCL** or **PYTHON** script processing.

reaction sqlshow

```
reaction sqlshow xhandle propertylist ?filterset? ?parameterdict?
x.sqlshow(property=,?mode=?,?parameters=?)
Reaction.Sqlshow(data,property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **`reaction get`** command. The differences between **`reaction get`** and **`reaction sqlshow`** are that the latter does not attempt computation of property data, but raises an error if the data is not present and valid, and that the **SQL** command variant formats the data as **SQL** values rather than for **Tcl** or **Python** script processing.

reaction subcommands

```
reaction subcommands
dir(Reaction)
```

Lists all subcommands of the **`reaction`** command. Note that this command does not require a reaction handle.

reaction swapin

```
reaction swapin xhandle
x.swapin()
```

Swap a reaction from the disk store fully back to memory, and disable further automatic loading and shelving. If the reaction was not swapped out, the command does nothing.

The command returns the original reaction handle or reference.

reaction swapout

```
reaction swapout xhandle
x.swapout()
```

Release most of the reaction data from memory and store it in a temporary disk store. The reaction handle remains valid. As soon as it is used in a command again after this command has been executed, the swapped reaction data is automatically reloaded from file, and then stored again when the object lock is released. To disable the automatic swapping of a reaction, use the **`reaction swapin`** command.

This command is intended to be used in cases where a large number of reactions must be kept in memory. Its use is not generally encouraged - it is only useful in case the programmer knows about access patterns. In other cases, the standard virtual memory mechanism of the operating system might yield better performance results.

The reactions are stored as binary blobs in a key/value store in a process-specific swap directory *cactvs%d*, (*%d* is replaced by the process ID) which is created automatically in the standard temporary directory. When a reaction is deleted, its swap record is also removed, if one was created during the lifetime of the reaction. When a **Cactvs** application program exits, the swap store as well as the swap directory are automatically deleted, even without explicit deletion of the last set of reactions in memory. In case of program crashes, the swap directory and its contents may however survive. If reaction swapping is used with unstable applications, the temporary directory should be checked from time to time.

The command returns the original reaction handle or reference.

Example:

```
rection swapout $xhandle
```

reaction tables

```
reaction tables xhandle ?filterlist?
x.tables(?filters=?)
```

Return a list of the handles or references of all table objects the reaction is associated with. Optionally, the table set may be filtered by a simple filter list. If the reaction is not related to any table, or none of these tables passes the filter list, an empty string is returned.

This command is only available if the toolkit was compiled with table support.

```
Example:
reaction tables $xhandle
```

reaction taint

```
reaction taint xhandle propertylist/changeset ?purge?
x.taint(property=,?purge=?)
```

Trigger a property data tainting event which acts on the reaction data, and the data of all ensembles in the reaction. If the reaction is a member of a dataset, or its ensembles are, the dataset and its objects are *not* tainted.

The command arguments are the same as for the **`ens taint`** command and explained there.

The command returns the original reaction handle or reference.

reaction transfer

```
reaction transfer xhandle propertylist ?targethandle? ?targetpropertylist?
x.transfer(properties=,?target=?,?targetproperties=?)
```

Copy property data from one reaction to another reaction or other major object, without going through an intermediate scripting language object representation, or dissociate property data from the reaction. If a property in the argument property list is not already valid on the source reaction, an attempt is made to compute it.

If a target object is specified, and a property is not a reaction property, the operation is implicitly performed on pairs of reaction ensembles with the same reaction role (property `E_REACTION_ROLE`) as if a **`ens transfer`** command were issued. In this command mode, the return value is the handle of the target reaction. The source and target reactions cannot be the same object.

If a target property list is given, the data from the source is stored as content of a different property on the target. For this, the data types of the properties must be compatible, and the object class of the target property that of the target object. No attempt is made to convert data of mismatched types. In case of multiple properties, the source property list and the target property list are stepped through in parallel. If there is no target property list, or it is shorter than the source list, unmatched entries are stored as original property values, and this implies that the object class of the source and target objects are the same.

If no target object is specified, or it is spelled as an empty string or **PYTHON `None`**, the visible effect of the command is the same as a simple **`reaction get`**, i.e. the result is the property data value or value list. The property data is then deleted from the source object. In case the data type of the deleted property was a major object (i.e. an ensemble, reaction, table, dataset or network), it is only unlinked from the source object, but not destroyed. This means that the object handles returned by the command can henceforth the used as independent objects. They can be deleted by a normal object deletion command, and are no longer managed by the source object.

Examples:

```
reaction transfer $xh X_IDENT $xh2
reaction transfer $xh A_MAPPING $xh2
```

The first example is a simple data copy. The second example transfers atom property `A_MAPPING` between the reagent and product ensembles of the reaction, and any other reaction ensemble where the property is valid, and a pair of ensembles with the same reaction role can be found. The order of the atoms in an ensemble pair is not required to be identical - property `A_LABEL` is used to identify corresponding atoms.

reaction trim

```
reaction trim xhandle ?propertylist?
x.trim(?properties=?)
```

Reduce the information content of a reaction to a standard minimum set and discard any additional information. This process minimizes the storage requirements of the reaction. The properties of the minimum set are computed if required. The retained property set is designed to support a faithful representation of the connectivity of the reaction ensembles including bond and atom labels and types as well as formal charges, stereochemistry, isotopes and atom mapping information, but not of any 2D or 3D coordinates or auxiliary additional attributes of atoms, bonds or other chemical objects or the reaction object proper.

The optional fourth argument is a list of properties which should be retained in addition to the standard set. If any of these are not present on the reaction (or its ensembles) that is to be trimmed, they are silently ignored and no attempt is made to compute them. Specifying properties of the standard retention set in this list is allowed but has no additional effect.

The return value of the command is a list of the remaining properties of the reaction and the reaction ensembles. The properties of the latter are reported as the *union* of the properties of the individual reaction ensembles (see **`reaction props`** command).

Example:

```
reaction trim $xhandle {X_SMILES E_NAME X_NAME}
```

reaction unlink

```
reaction unlink xhandle ?xhandle/all?
x.unlink("all")
x.unlink(?xhandle/xref?,...)
```

Remove links of the reaction to other reactions. The special string argument *all* removes all links of the reaction. If an argument is a reaction the first reaction is not linked to, it is ignored.

The command returns the number of severed reaction links.

reaction unlock

```
reaction unlock xhandle propertylist/reaction/all
x.unlock(property=)
```

Unlock property data for the reaction, meaning that they are again under the control of the standard data consistency manager.

The property data to unlock can be selected by providing a list of the following identifiers:

- Property names or references
  Valid property instances on the reaction, reaction ensembles, or ensemble minor objects are unlocked. Non-existent data is silently ignored. It is not possible to unlock individual property fields.
- *all*
  All valid reaction properties are unlocked. Ensemble properties and ensemble minor object properties are not affected.
- *reaction*
  This is an object class identifier. All property data which is controlled by the reaction major object and attached to the specified object class is unlocked. Since reactions do not contain minor objects, this identifier is equivalent to *all*.

Property data locks are obtained by the **`reaction lock`** command.

The return value is the original reaction handle or reference.

Example:

```
set xhandle [reaction create {C=C>[Pt]>CC}]
reaction lock $xhandle X_GIF 1
reaction clear $xhandle agent
reaction unlock $xhandle X_GIF
```

In this example, first a reaction depiction in property data `X_GIF` is generated and locked. After that, the reaction and reaction ensembles can be manipulated without losing the image data. The agent is removed in the next step - but the `X_GIF` image which shows the catalyst and which normally would have been deleted when the agent is removed is kept. Finally, the image property data is put back under the standard control of the data consistency manager.

reaction unpack

```
reaction unpack packstring ?compressionlib?
Reaction.Unpack(data=,?compressionlib=?)
```

Unpack a base64-encoded serialized object string which was created by a **`reaction pack`** command. The return value of this function is the handle or reference of the newly created reaction object, which is an exact duplicate of the packed original reaction.

Reactions may also be unpacked by a **`reaction create`** command.

The default compression library is *zlib*. For more options, see **`reaction pack`**.

Example:

```
set packdata [reaction pack [reaction create C=O>>CO]]

set xhandle [reaction unpack $packdata]
```

reaction valid

```
reaction valid xhandle propertylist
```

```
x.valid(property/propertysequence)
```

Returns a list of boolean values indicating whether values for the named properties are currently set for the reaction. No attempt at computation is made. For **Python**, where single-item lists are syntactically not the same as a single value, the return value is a single boolean if the argument was a string or a property reference, and only a single property was decoded.

Example:

```
reaction valid $xhandle X_IDENT
```

reports whether the reaction has a standard ID (has a valid `X_IDENT` property) or not.

`reaction has` is an alias to this command.

reaction verify

```
reaction verify xhandle property
x.verify(property)
```

Verify the values of the specified property on the reaction. The property data must be valid, and a reaction property. If the data can be found, it is checked against all constraints defined for the property, and, if such a function has been defined, is tested with the value verification function of the property.

If all tests are passed, the return value is boolean 1, 0 if the data could be found but fails the tests, and an error condition otherwise.

reaction weed

```
reaction weed xhandle keywords
x.weed(keywordsequence)
x.weed(?keyword?,...)
```

This command performs standard clean-up operations on all ensembles in the reaction. The supported operations are described in more detail in the section on the equivalent **`ens weed`** command.

The return value of this command is the original reaction handle or reference.

reaction xdelete

```
reaction xdelete ?xhandle?...
reaction xdelete all
x.xdelete()
Reaction.Xdelete("all")
Reaction.XDelete(?xrefsequence/xref/xhandle?,...)
```

Delete reactions and the ensembles which are part of the deleted reactions. The special parameter *all* may be used to delete all reactions currently registered in the application. Alternatively, any number of lists of reaction handles may be specified for specific reaction deletions. If a reaction is part of a linked reaction network, all reactions in the network are also deleted. This is the difference to the normal *delete* command or method.

The command returns the number of deleted reactions.

reaction xdup

```
reaction xdup xhandle ?dataset? ?position?
x.xdup(?target=?,?position=?)
```

Duplicate the specified reaction and its ensembles, together with any additional reactions it may be linked to (see **`reaction link/unlink`**). The latter behavior is the difference to the standard **`reaction dup`** command. The return value is the new reaction handle or reference. If additional reactions were duplicated as part of a linked networks, these can be queried via **`reaction links`**. The duplicated reaction ensembles are also assigned unique handles.

The duplicate ensemble is placed into the same dataset as the source, if it is a member of a dataset. Specifying an explicitly empty dataset argument (including **`None`** for **Python**) places the duplicate outside any dataset, regardless of the dataset membership of the source ensemble.

If the duplicate is moved to a dataset, it is appended to the dataset end by default. This happens also if the position parameter is explicitly specified as *end* or an empty string. Otherwise, the reaction is inserted at the given position, starting with 0. If the requested position is larger than the current size of the dataset, the reaction is appended.

The *ring* Command

The *ring* command is the generic command used to manipulate rings. The syntax of this command follows the standard schema of *command/subcommand/majorhandle/minorlabel*.

Pseudo ring labels *first*, *last* and *random* are special values, which select the first ring in the ring list, the last, or a random ring.

Examples:

```
ring get $ehandle 1 R_SIZE
```

This is the list of officially supported subcommands:

ring append

```
ring append ehandle label ?property value?...
r.append({?property:value,?...})
r.append(?property,value,?...)
```

Standard data manipulation command for appending property data. It is explained in more detail in the section about setting property data.

The command returns the first data value.

Example:

```
ring append $ehandle 1 R_NAME "_centroid"
```

ring atoms

```
ring atoms ehandle label ?filterset? ?filtermode?
r.atoms(?filters=?,?mode=?)
```

Standard cross-referencing command to obtain the labels or references of the atom in the ring. This is explained in more detail in the section about object cross-references.

Example:

```
ring atoms $ehandle 1 carbon
```

returns the labels of the carbon atoms in the ring.

ring bonds

```
ring bonds ehandle label ?filterset? ?filtermode?
r.bonds(?filters=?,?mode=?)
```

Standard cross-referencing command to obtain the labels or references of the bonds the ring contains. This is explained in more detail in the section about object cross-references. Technically, a ring contains atoms, not bonds. This command lists all bonds which exist between consecutive atoms in the ring and which are of a type which matches the current ring bond mask.

Examples:

```
ring bonds $ehandle 1
ring bonds $ehandle 1 {1 doublebond triplebond} count
```

The first example returns all labels of the bonds ring 1 contains. The second example returns the number of double or triple bonds in the ring.

ring defined

```
ring defined ehandle label property
r.defined(property)
```

This command checks whether a property is defined for the ring. This is explained in more detail in the section about property validity checking. Note that this is *not* a check for the presence of property data! The **`ens valid`** command is used for this purpose.

Example:

```
ring defined $ehandle 1 R_AROMATIC
```

checks whether ring 1 is of a type for which property `R_AROMATIC` is defined.

ring delete

```
ring delete ehandle ?label?...
ring delete ehandle all
r.delete()
Ring.Delete(eref,"all")
Ring.Delete(rref,...)
Ring.Delete(eref,?rlabel/rref/rrefsequence?,...)
```

This command removes rings from the ensemble ring list and destroys them. A *ring* property invalidation event is generated and thus the command may indirectly change the ensemble data.

This command is rarely used. Rings are usually generated and destroyed automatically.

The command returns the number of deleted items.

ring dget

```
ring dget ehandle label propertylist ?filterset? ?parameterdict?
r.dget(property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **`ring get`** command. The difference between **`ring get`** and **`ring dget`** is that the latter does not attempt computation of property data, but rather initializes the property values to the default and return that default if the data is not yet available. For data already present, **`ring get`** and **`ring dget`** are equivalent.

ring ens

```
r.ens()
```

PYTHON-only method to get the ensemble reference from a ring reference.

ring exists

```
ring exists ehandle label ?filterlist?
r.exists(?filters=?)
Ring.Exists(eref,label,?filters=?)
```

Check whether this ring exists. Optionally, a filter list can be supplied to check for the presence of specific features. The command returns boolean 0 if the ring does not exist, or fails the filter, and 1 in case of successful testing.

Example:

```
ring exists $ehandle 99
```

ring expr

```
ring expr ehandle label expression
r.expr(expression)
```

Compute a standard **SQL**-style property expression for the ring. This is explained in detail in the chapter on property expressions.

ring fill

```
ring fill ehandle label ?property value?...
r.fill({property:value,...})
r.fill(?property,value?,...)
```

Standard data manipulation command for setting data, ignoring possible mismatches between the lengths of the lists of objects associated with the property and the value list. It is explained in more detail in the section about setting property data.

The command returns the first fill value.

Example:

```
ring fill $ehandle 1 B_COLOR red
```

sets the color of the first bond ring 1 contains to *red*.

ring filter

```
ring filter ehandle label filterlist
r.filter(filters)
```

Check whether a ring passes a filter list. The return value is boolean 1 for success and 0 for failure.

Example:

```
ring filter $ehandle 1 [list carbon doublebond]
```

checks whether the ring contains one or more carbon atoms and one or more double bonds. The double bond does not need to contain a carbon atom.

ring formulamatch

```
ring formulamatch ehandle label formula_expression ?other_elements?
r.formulamatch(query=,?other_elements=?)
```

Match the ring against a formula expression. Its syntax is the same as in formula queries in **`molfile scan`** and other scan commands.

There are several methods to specify whether any elements not mentioned in the formula expression may or must be present. If the *other_elements* flag is used, it has the highest priority. If may be set to 0 (no other elements allowed), 1 (allowed) or 2 (required), and if it is set, any prefix in the formula expression is ignored. If it is not used, a prefix in the formula expression may be used to control the matching. Supported prefixes are = (no other elements), >= (other elements allowed) and > (required). If no prefix is used, the default mode is an exact match without other elements.

The return value is the boolean match result.

Example:

```
ring formulamatch $eh 1 >C6
```

Tests whether the ring contains six carbon atoms. At least one atom which is not carbon must be present.

```
ring formulamatch $eh 1 C5-6(Cl+Br+I)2- 1
```

Tests whether the ring has five or six carbon atoms, two ore more heavy halogens, and potentially any other elements.

ring get

```
ring get ehandle label propertylist ?filterset? ?parameterdict?
r.get(property=,?filters=?,?parameters=?)
r[property]
r.property
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

Examples:

```
ring get $ehandle 1 {R_SIZE A_ELEMENT}
```

yields a list with two elements, consisting of the ring size as the first element and the element numbers of all atoms in the ring as a nested list as the second result list element. If the information is not yet available, an attempt is made to compute it. If the computation fails, an error results.

```
ring get $ehandle 1 B_ORDER cxbond
```

reports the bond orders of all bonds of the ring which are carbon-heteroatom bonds.

For the use of the optional property parameter list argument, refer to the documentation of the **`ens get`** command.

Variants of the **`ring get`** command are **`ring dget, ring new, ring nget, ring show, ring sqldget, ring sqlget, ring sqlnew,`** and **`ring sqlshow`**.

Further examples:

```
ring get $ehandle 1 E_NAME
ring get $ehandle 1 A_FLAGS(boxed)
```

ring groups

```
ring groups ehandle label ?filterset? ?filtermode?
r.groups(?filters=?,?mode=?)
```

Standard cross-referencing command to obtain the labels or references of the groups the ring overlaps with. This is explained in more detail in the section about object cross-references. An overlap between a ring and a group is established when there are common atoms which are contained in both objects.

Example:

```
ring groups $ehandle 1
```

ring hydrogenate

```
ring hydrogenate ehandle label ?filterset? ?changeset?
```

```
r.hydrogenate(?filters=?,?changeset=?)
```

Reduce all bonds in the ring to single bonds except those excluded by the filter set.

If a change set is supplied, its interpretation is the same as in **`mol hadd`**.

The command returns the number of added hydrogens.

Example:

```
ring hydrogenate $eh 1 {!arobond !ccbond}
```

This reduces all non-aromatic hetero bonds in ring 1 to single bonds.

ring index

```
ring index ehandle label
r.index()
```

Get the index of the ring. The index is the position in the ring list of the ensemble. The first position is index 0.

Example:

```
ring index $ehandle 99
```

ring jget

```
ring jget ehandle label propertylist ?filterset? ?parameterdict?
r.jget(property=,?filters=?,?parameters=?)
```

This is a variant of **`ring get`** which returns the result data as a **JSON** formatted string instead of **Tcl** interpreter objects.

ring jnew

```
ring jnew ehandle label propertylist ?filterset? ?parameterdict?
g.jnew(property=,?filters=?,?parameters=?)
```

This is a variant of **`ring new`** which returns the result data as a **JSON** formatted string instead of **Tcl** interpreter objects.

ring jshow

```
ring jshow ehandle label propertylist ?filterset? ?parameterdict?
g.jshow(property=,?filters=?,?parameters=?)
```

This is a variant of **`ring show`** which returns the result data as a **JSON** formatted string instead of **Tcl** interpreter objects.

ring ligands

```
ring ligands ehandle label ?filterset? ?filtermode? ?sphere?
r.ligands(?filters=?,?mode=?,?sphere=?)
```

Get the labels of atoms that are ligands to the current ring, i.e. they are not member of the ring, but bonded to it. The *filterset* and *filtermode* parameters work as with other object cross-reference commands.

This command supports a special *filtermode* parameter in addition to the standard set (*exists*, *count*, *exclude*, *include*). The *bonds* parameter, followed by a bit set combination from the allowed values *ring*, *sidechain* or *bridge* can be used for topological filtering of the traversable bonds. By default, no topological bond filtering is applied.

The default sphere number is one, meaning only the direct ring substituents are reported. A higher positive number extracts more distant atoms, but still excludes the ring atoms. A negative sphere number works the same way, but the result set includes atoms seen in inner spheres. Duplicate atoms encountered in different spheres are not reported by default.

If the filter set contains a bond filter, it is applied to the bond linking the first sphere atom to the ring or leading from an inner to the outer sphere atom. This means it is not sufficient for an atom to possess any bond which passes the filter, but it must be the bond to the ring or sphere expansion atom.

Example:

```
set nonringsubcnt [rings ligands $eh $rlabel {!hydrogen !ringatom} count]
```

ring local

```
ring local ehandle label propertylist ?filterset? ?parameterdict?
r.local(property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading and recalculating object data. It is explained in more detail in the section about retrieving property data.

Example:

```
ring local $ehandle 1 A_LABEL_STEREO
```

Note that very few computation routines currently support the local re-computation of data - in most cases, this command falls back to a global re-computation.

ring match

```
ring match ehandle label ss_ehandle ?ss_label? ?matchflags? ?ignoreflags? ?
   atommatchvar? ?bondmatchvar? ?molmatchvar?
r.match(substructure=,?substructurering=?,?matchflags=?,?ignoreflags=?,
   ?atommatchvariable=?,?bondmatchvariable=?,?molmatchvariable=?)
```

Check whether the selected ring matches a substructure. Only the first substructure ring, or the ring selected by the substructure label parameter, is tested. The substructure may be part of any structure ensemble, and even be in the same ensemble as the primary command ring. Both the atoms in the ring and the bonds between them are checked.

The precise operation of the substructure match routine can be tuned by providing a standard set of match flags and feature ignore flags. The default match flag set has set bits for the *bondorder*, *atomtree* and *bondtree* comparison features, and an empty ignore set. If a flag set is specified as an empty string, the default set is used. In order to reset a flag set, an explicit *none* value must be used.

The command returns 1 for a successful match, 0 otherwise. If an optional atom, bond, or molecule map variable is specified, it is set to a nested list of matching substructure/structure atom, bond or molecule labels. If no match can be found, the variable is set to an empty list. In case only a bond or molecule map variable is needed, an empty string can be used to skip the unused map variable argument positions.

Example:

```
set ss [ens create {c1ccccc1} smarts]
set r_is_phenyl [ring match $ehandle $label $ss]
```

ring mols

```
ring mols ehandle label ?filterset? ?filtermode?
r.mols(?filters=?,?mode=?)
```

Standard cross-referencing command to obtain the labels or references of the molecules the ring is contained in. This is explained in more detail in the section about object cross-references. Under specific circumstances, it is possible to have rings which span more than one molecule.

Examples:

```
ring mols $ehandle 1
ring mols $ehandle 1 [list heterocycle aroring]
```

The first example returns the labels of all molecules the ring is a part of. The second example filters the molecules - only molecules which contain heteroaromatic rings are reported. The ring filter is applied to the molecule because this is the return object, not the ring, so this filter does not require the ring the command was issued for to be in that class.

ring new

```
ring new ehandle label propertylist ?filterset? ?parameterdict?
r.new(property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **`ring get`** command. The difference between **`ring get`** and **`ring new`** is that the latter forces the re-computation of the property data, regardless whether it is present and valid, or not.

ring nget

```
ring nget ehandle label propertylist ?filterset? ?parameterdict?
r.nget(property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **`ring get`** command. The difference between **`ring get`** and **`ring nget`** is that the latter always returns numeric data, even if symbolic names for the values are available.

ring pis

```
ring pis ehandle label ?filterset? ?filtermode?
r.pis(?filter=?,?mode=?)
```

Standard cross-referencing command to obtain the labels or references of the π systems the ring overlaps with. This is explained in more detail in the section about object cross-references.

Examples:

```
ring pis $ehandle 1
```

π systems are a rather exotic feature and not commonly used. These are essentially descriptions of bonding interactions which use p or d orbitals, such as in standard covalent multiple bonds. A simple double bond is described with one σ system and one π system in this representation.

ring ref

```
Ring.Ref(eref,identifier)
```

Python only method to get a ring reference. See **`ring ring`** command.

ring ring

```
ring ring ehandle label
Ring.Ref(eref,identifier)
```

Standard cross-referencing command to obtain the label of the ring as stored in property `R_LABEL`. This is explained in more detail in the section about object cross-references.

Example:

```
ring ring $ehandle #0
```

returns the label of the first ring of the ensemble ring list.

ring ringsystem

```
ring ringsystem ehandle label ?filterset? ?filtermode?
r.ringsystem(?filters=?,?mode=?)
```

Standard cross-referencing command to obtain the labels or references of the ring system the ring is a member of. This is explained in more detail in the section about object cross-references.

Examples:

```
ring ringsystem $ehandle 1
ring ringsystem $ehandle 1 [list heterocycle aroring]
```

The first example returns the label of the ring system the ring is a member of. The second example filters the ring system - a ring system label is obtained only if that ring system contains one or more hetero aromats. These filters are applied to the ring system, meaning that they are implicitly applied to all rings in the ring system, not just the ring used for the query command.

Since a ring can only be a member of a single ring system, the command spells the target in singular.

ring set

```
ring set ehandle label ?property value?...
r.set(?property,value?,...)
r.set({property:value,...})
r.property = value
r[property] = value
```

Standard data manipulation command. It is explained in more detail in the section about setting property data.

Example:

```
ring set $ehandle 1 R_NAME "The central pharmacophore"
```

ring show

```
ring show ehandle label propertylist ?filterset? ?parameterdict?
r.show(property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **`ring get`** command. The difference between **`ring get`** and **`ring show`** is that the latter does not attempt computation of property data, but raises an error if the data is not present and valid. For data already present, **`ring get`** and **`ring show`** are equivalent.

ring sigmas

```
ring sigmas ehandle label ?filterset? ?filtermode?
r.sigmas(?filters=?,?mode=?)
```

Standard cross-referencing command to obtain the labels or references of the σ systems the ring overlaps with. This is explained in more detail in the section about object cross-references.

Examples:

```
ring sigmas $ehandle 1
```

σ systems are a rather exotic feature and not commonly used. These are essentially descriptions of bonding interactions which use s orbitals, such as normal, covalent single bonds, or the central bond in multiple bonds. A simple double bond is described with one σ system and one π system in this representation.

ring sqldget

```
ring sqldget ehandle label propertylist ?filterset? ?parameterdict?
r.sqldget(property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **`ring get`** command. The differences between **`ring get`** and **`ring sqldget`** are that the latter does not attempt computation of property data, but initializes the property value to the default and returns that default, if the data is not present and valid; and that the **SQL** command variant formats the data as **SQL** values rather than for **Tcl** or **Python** script processing.

ring sqlget

```
ring sqlget ehandle label propertylist ?filterset? ?parameterdict?
r.sqldget(property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **`ring get`** command. The difference between **`ring get`** and **`ring sqlget`** is that the **SQL** command variant formats the data as **SQL** values rather than for **Tcl** or **Python** script processing.

ring sqlnew

```
ring sqlnew ehandle label propertylist ?filterset? ?parameterdict?
r.sqldget(property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **`ring get`** command. The differences between **`ring get`** and **`ring sqlnew`** are that the latter forces re-computation of the property data, and that the **SQL** command variant formats the data as **SQL** values rather than for **TCL** or **PYTHON** script processing.

ring sqlshow

```
ring sqlshow ehandle label propertylist ?filterset? ?parameterdict?
r.sqlshow(property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **`ring get`** command. The differences between **`ring get`** and **`ring sqlshow`** are that the latter does not attempt computation of property data, but raises an error if the data is not present and valid, and that the **SQL** command variant formats the data as **SQL** values rather than for **TCL** or **PYTHON** script processing.

ring subcommands

```
ring subcommands
dir(Ring)
```

Lists all subcommands of the **`ring`** command. Note that this command does not require an ensemble handle, or a label.

ring surfaces

```
ring surfaces ehandle label ?filterset? ?filtermode?
r.surfaces(?filters=?,?mode=?)
```

Standard cross-referencing command to obtain the labels or references of surface patches the ring is associated with. This is explained in more detail in the section about object cross-references.

Example:

```
ring surfaces $ehandle $label
```

Note that surface patches do not need to be associated with an atom, and if they are not, they are implicitly not associated with any ring.

ring xbonds

```
ring xbonds ehandle label ?filterset? ?filtermode?
r.xbonds(?filters=?,?mode=?)
```

Get labels or references of crossing bonds which are not contained in the ring (defined as all bonds between any two consecutive registered atoms in the system), but have one atom in the ring. Bonds between atoms in the ring which are not part of the ring bond set are reported. For example, when looking at the six-membered outer envelope ring of bicyclobutane, the bridge between the two four-membered SSSR bonds is included in the result set.

The *ringsystem* Command

The *ringsystem* command is the generic command used to manipulate ring systems. The syntax of this command follows the standard schema of *command/subcommand/majorhandle/minorlabel*.

Pseudo ring system labels *first*, *last* and *random* are special values, which select the first ring system in the ring system list, the last, or a random ring system.

Examples:

```
ring get $ehandle 1 Y_NATOMS
```

The command can also be invoked as shortened alias **`ringsys`**.

The default property name prefix for ringsystem-level properties is `Y_`. The `R_` prefix is used for more commonly encountered ring properties.

This is the list of officially supported subcommands:

ringsystem append

```
ringsystem append ehandle label ?property value?...
y.append({?property:value,?...})
y.append(?property,value,?...)
```

Standard data manipulation command for appending property data. It is explained in more detail in the section about setting property data.

The command returns the first data value.

Example:

```
ringsystem append $ehandle 1 Y_NAME "_centroid"
```

ringsystem atoms

```
ringsystem atoms ehandle label ?filterset? ?filtermode?
y.atoms(?filters=?,?mode=?)
```

Standard cross-referencing command to obtain the labels or references of the atom in the ring system. This is explained in more detail in the section about object cross-references.

Example:

```
ringsystem atoms $ehandle carbon
```

returns the labels of the carbon atoms in the ring system.

ringsystem bonds

```
ringsystem bonds ehandle label ?filterset? ?filtermode?
y.bonds(?filters=?,?mode=?)
```

Standard cross-referencing command to obtain the labels or references of the bonds the ring system contains. This is explained in more detail in the section about object cross-references.

Technically, a ring system contains atoms, not bonds. This command lists all bonds which exist between atoms in the ring system and which are of a type which matches the current ring bond mask and which are ring bonds.

Examples:

```
rinsysg bonds $ehandle 1
ringsystem bonds $ehandle 1 {1 doublebond triplebond} count
```

The first example returns all labels of the bonds ring system 1 contains. The second example returns the number of double or triple bonds in the ring system.

ringsystem create

```
ringsystem create ehandle ?atom/atomlist?...
Ringsystem(eref,?aref/arefsequence/alabel?,...)
Ringsystem(aref,...)
Ringsystem.Create(eref,?aref/arefsequence/alabel?,...)
Ringsystem.Create(aref,...)
```

This command can be used to manually create a ring system or pseudo ring system from an arbitrary collection of atoms. No check is made whether the atoms actually form a valid ring system. The result value of the command is the label of the newly created ring system. This command generates a *ringsystem* property invalidation event and may thus indirectly influence the ensemble data.

By default, ring systems are automatically created whenever they are referenced and not yet set up for the context ensemble.

The command returns the label or reference of the new ringsystem.

Example:

```
ringsystem create $ehandle {1 2 3}
```

ringsystem defined

```
ringsystem defined ehandle label property
y.defined(property)
```

This command checks whether a property is defined for the ring system. This is explained in more detail in the section about property validity checking. Note that this is *not* a check for the presence of property data! The **`ens valid`** command is used for this purpose.

Example:

```
ringsystem defined $ehandle 1 Y_NAME
```

checks whether ring system 1 is of a type for which property `Y_NAME` is defined.

ringsystem delete

```
ringsystem delete ehandle ?label?...
ringsystem delete ehandle all
y.delete()
Ringsystem.Delete(eref,"all")
Ringsystem.Delete(yref,...)
Ringsystem.Delete(eref,?ylabel/yref/yrefsequence?,...)
```

This command removes ring systems from the ensemble ring system list and destroys them. A *ringsystem* property invalidation event is generated and thus the command may indirectly change the ensemble data.

The command returns the number of deleted items.

This command is rarely used. Ring systems are usually generated and destroyed automatically.

ringsystem dget

```
ringsystem dget ehandle label propertylist ?filterset? ?parameterdict?
y.dget(property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **`ringsystem get`** command. The difference between **`ringsystem get`** and **`ringsystem dget`** is that the latter does not attempt computation of property data, but rather initializes the property values to the default and return that default if the data is not yet available. For data already present, **`ringsystem get`** and **`ringsystem dget`** are equivalent.

ringsystem ens

```
y.ens()
```

Python-only method to get the ensemble reference from a ringsystem reference.

ringsystem exists

```
ringsystem exists ehandle label ?filterlist?
r.exists(?filters=?)
Ringsystem.Exists(eref,label,?filters=?)
```

Check whether this ring system exists. Optionally, a filter list can be supplied to check for the presence of specific features. The command returns boolean 0 if the ring system does not exist, or fails the filter, and 1 in case of successful testing.

Example:

```
ringsystem exists $ehandle 99
```

ringsystem expr

```
ringsystem expr ehandle label expression
y.exp(rexpression)
```

Compute a standard **SQL**-style property expression for the ring system. This is explained in detail in the chapter on property expressions.

ringsystem fill

```
ringsystem fill ehandle label ?property value?...
y.fill({property:value,...})
y.fill(?property,value?,...)
```

Standard data manipulation command for setting data, ignoring possible mismatches between the lengths of the lists of objects associated with the property and the value list. It is explained in more detail in the section about setting property data.

The command returns the first fill value.

Example:

```
ringsystem fill $eh 1 A_COLOR [lrepeat [ringsystem atoms $ehandle 1 {} count] red
```

sets the color of the atoms in ring system 1 to *red*.

ringsystem filter

```
ringsystem filter ehandle label filterlist
y.filter(filters)
```

Check whether a ring system passes a filter list. The return value is boolean 1 for success and 0 for failure.

Example:

```
ringsystem filter $ehandle 1 [list carbon doublebond]
```

checks whether the ring system contains one or more carbon atoms and one or more double bonds. The double bond does not need to include a carbon atom.

ringsystem formulamatch

```
ringsystem formulamatch ehandle label formula_expression ?other_elements?
y.formulamatch(query=,?other_elements=?)
```

Match the ringsystem against a formula expression. Its syntax is the same as in formula queries in **`molfile scan`** and other scan commands.

There are several methods to specify whether any elements not mentioned in the formula expression may or must be present. If the *other_elements* flag is used, it has the highest priority. If may be set to 0 (no other elements allowed), 1 (allowed) or 2 (required), and if it is set, any prefix in the formula expression is ignored. If it is not used, a prefix in the formula expression may be used to control the matching. Supported prefixes are = (no other elements), >= (other elements allowed) and > (required). If no prefix is used, the default mode is an exact match without other elements.

The return value is the boolean match result.

Examples:

```
ringsystem formulamatch $eh 1 >C6
```

Tests whether the ring system contains six carbon atoms. At least one atom which is not carbon must be present.

```
ringsystem formulamatch $eh 1 C5-6(Cl+Br+I)2- 1
```

Tests whether the ring system has five or six carbon atoms, two ore more heavy halogens, and potentially any other elements.

ringsystem get

```
ringsystem get ehandle label propertylist ?filterset? ?parameterdict?
y.get(property=,?filters=?,?parameters=?)
y[property]
y.property
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

Examples:

```
ringsystem get $ehandle 1 {Y_NATOMS A_ELEMENT}
```

yields a list with two elements, consisting of the ring system size as the first element and the element numbers of all atoms in the ring system as a nested list as the second result list element. If the information is not yet available, an attempt is made to compute it. If the computation fails, an error results.

```
ringsystem get $ehandle 1 B_ORDER cxbond
```

reports the bond orders of all bonds of the ring system which are carbon-hetero bonds.

For the use of the optional property parameter list argument, refer to the documentation of the **ens get** command.

Variants of the **ringsystem get** command are **ringsystem dget, ringsystem new, ringsystem nget, ringsystem show, ringsystem sqldget, ringsystem sqlget, ringsystem sqlnew,** and **ringsystem sqlshow.**

Further examples:

```
ringsystem get $ehandle 1 E_NAME
ringsystem get $ehandle 1 A_FLAGS(boxed)
```

ringsystem groups

```
ringsystem groups ehandle label ?filterset? ?filtermode?
y.groups(?filters=?,?mode=?)
```

Standard cross-referencing command to obtain the labels or references of the groups the ring system overlaps with. This is explained in more detail in the section about object cross-references. An overlap between a ring system and a group is established when there are common atoms which are contained in both objects.

Example:

```
ringsystem groups $ehandle 1
```

ringsystem hydrogenate

```
ringsystem hydrogenate ehandle label ?filterset? ?changeset?
y.hydrogenate(?filters=?,?changeset=?)
```

Reduce all bonds in the ringsystem to single bonds except those excluded by the filter set.

If a change set is supplied, its interpretation is the same as in **mol hadd.**

The command returns the number of added hydrogens.

Example:

```
ringsystem hydrogenate $eh 1 {!arobond !ccbond}
```

This reduces all non-aromatic hetero bonds in ringsystem 1 to single bonds.

ringsystem index

```
ringsystem index ehandle label
y.index()
```

Get the index of the ring system. The index is the position in the ring system list of the ensemble. The first position is index 0.

Example:

```
ringsystem index $ehandle 99
```

ringsystem jget

```
ringsystem jget ehandle label propertylist ?filterset? ?parameterdict?
y.jget(property=,?filters=?,?parameters=?)
```

This is a variant of **`ringsystem get`** which returns the result data as a **JSON** formatted string instead of **TCL** or **PYTHON** interpreter objects.

ringsystem jnew

```
ringsystem jnew ehandle label propertylist ?filterset? ?parameterdict?
y.jnew(property=,?filters=?,?parameters=?)
```

This is a variant of **`ringsystem new`** which returns the result data as a **JSON** formatted string instead of **TCL** or **PYTHON** interpreter objects.

ringsystem jshow

```
ringsystem jshow ehandle label propertylist ?filterset? ?parameterdict?
y.jshow(property=,?filters=?,?parameters=?)
```

This is a variant of **`ringsystem show`** which returns the result data as a **JSON** formatted string instead of **TCL** or **PYTHON** interpreter objects.

ringsystem ligands

```
ringsystem ligands ehandle label ?filterset? ?filtermode? ?sphere?
y.ligands(?filters=?,?mode=?,?sphere=?)
```

Get the labels of atoms that are ligands to the current ring system, i.e. they are not member of the ring system, but bonded to it. The *filterset* and *filtermode* parameters work as with other object cross-reference commands.

This command supports a special *filtermode* parameter in addition to the standard set (*exists*, *count*, *exclude*, *include*). The *bonds* parameter, followed by a bit set combination from the allowed values *ring*, *sidechain* or *bridge* can be used for topological filtering of the traversable bonds. By default, no topological bond filtering is applied.

The default sphere number is one, meaning only the direct ring system substituents are reported. A higher positive number extracts more distant atoms, but still excludes the ring system atoms. A negative sphere number works the same way, but the result set includes atoms seen in inner spheres. Duplicate atoms encountered in different spheres are not reported by default.

If the filter set contains a bond filter, it is applied to the bond linking the first sphere atom to the ring system or leading from an inner to the outer sphere atom. This means it is not sufficient for an atom to possess any bond which passes the filter, but it must be the bond to the ring system or sphere expansion atom.

Example:

```
set subcnt [ringsystem ligands $eh $ylabel {!hydrogen} count]
```

ringsystem local

```
ringsystem local ehandle label propertylist ?filterset? ?parameterdict?
y.local(property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading and recalculating object data. It is explained in more detail in the section about retrieving property data.

Example:

```
ringsystem local $ehandle 1 A_LABEL_STEREO
```

Note that very few computation routines currently support the local re-computation of data - in most cases, this command falls back to a global re-computation.

ringsystem match

```
ringsystem match ehandle label ssehandle ?sslabel? ?matchflags? ?ignoreflags?
   ?atommatchvar? ?bondmatchvar? ?molmatchvar?
```

```
y.match(substructure=,?substructureringsystem=?,?matchflags=?,?ignoreflags=?,
   ?atommatchvariable=?,?bondmatchvariable=?,?molmatchvariable=?)
```

Check whether the selected ringsystem matches a substructure. Only the first substructure ring system, or the ring system selected by the substructure label parameter, is tested. The substructure may be part of any structure ensemble, and even be in the same ensemble as the primary command ring system. Both the atoms in the ringsystem and the bonds between them are checked.

The precise operation of the substructure match routine can be tuned by providing a standard set of match flags and feature ignore flags. The default match flag set has set bits for the *bondorder*, *atomtree* and *bondtree* comparison features, and an empty ignore set. If a flag set is specified as an empty string, the default set is used. In order to reset a flag set, an explicit *none* value must be used.

The command returns 1 for a successful match, 0 otherwise. If an optional atom, bond, or molecule map variable is specified, it is set to a nested list of matching substructure/structure atom, bond or molecule labels. If no match can be found, the variable is set to an empty list. In case only a bond or molecule map variable is needed, an empty string can be used to skip the unused map variable argument positions.

Example:

```
set ss [ens create {c1ccccc1.c1ncccc1} smarts]
set rs_contains_phenyl [ringsystem match $ehandle $label $ss 1]
```

ringsystem mols

```
ringsystem mols ehandle label ?filterset? ?filtermode?
```

```
y.mols(?filters=?,?mode=?)
```

Standard cross-referencing command to obtain the labels or references of the molecules the ring system participates in. This is explained in more detail in the section about object cross-references. Under specific circumstances, it is possible to have ring systems which span more than one molecule.

Examples:

```
ringsystem mols $ehandle 1
ringsystem mols $ehandle 1 [list heterocycle aroring]
```

The first example returns the labels of all molecules the ring system is a part of. The second example filters the molecules - only molecules which contain heteroaromatic rings are reported. The ring filter is applied to the molecule because this is the return object, not the ring, so this filter does not require the ring the command was issued for to be in that class.

ringsystem new

```
ringsystem new ehandle label propertylist ?filterset? ?parameterdict?
y.new(property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **`ringsystem get`** command. The difference between **`ringsystem get`** and **`ringsystem new`** is that the latter forces the re-computation of the property data, regardless whether it is present and valid, or not.

ringsystem nget

```
ringsystem nget ehandle label propertylist ?filterset? ?parameterdict?
y.new(property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **`ringsystem get`** command. The difference between **`ringsystem get`** and **`ringsystem nget`** is that the latter always returns numeric data, even if symbolic names for the values are available.

ringsystem pis

```
ringsystem pis ehandle label ?filterset? ?filtermode?
y.pis(?filters=?,?mode=?)
```

Standard cross-referencing command to obtain the labels or references of the π systems the ring system overlaps with. This is explained in more detail in the section about object cross-references.

Examples:

```
ringsystem pis $ehandle 1
```

π systems are a rather exotic feature and not commonly used. These are essentially descriptions of bonding interactions which use p or d orbitals, such as in standard covalent multiple bonds. A simple double bond is described with one σ system and one π system in this representation.

ringsystem ref

```
Ringsystem.Ref(eref,identifier)
```

PYTHON only method to get a ringsystem reference. See **`ringsystem ringsystem`** command.

ringsystem rings

```
ringsystem rings ehandle label ?filterset? ?filtermode?
y.rings(?filters=?,?mode=?)
```

Standard cross-referencing command to obtain the labels or references of the ring system the ring is a member of. This is explained in more detail in the section about object cross-references.

Examples:

```
ringsystem rings $ehandle 1
ringsystem rings $ehandle 1 [list heterocycle aroring]
```

The first example returns the labels of the rings which are contained in the ring system. The second example filters the ring system - a ring label is obtained only if that ring system contains one or more heteroaromatic rings. These filters are applied to the individual rings in the ring system, not the command ringsystem.

ringsystem ringsystem

```
ringsystem ringsystem ehandle label
Ringsystem.Ref(eref,identifier)
```

Standard cross-referencing command to obtain the label or reference of the ring. This is explained in more detail in the section about object cross-references.

Example:

```
ringsystem ringsystem $ehandle #0
```

returns the label of the first ring system of the ensemble ring system list.

ringsystem set

```
ringsystem set ehandle label ?property value?...
y.set(?property,value?,...)
y.set({property:value,...})
y.property = value
y[property] = value
```

Standard data manipulation command. It is explained in more detail in the section about setting property data.

Example:

```
ringsystem set $ehandle 1 Y_NAME "The central pharmacophore"
```

ringsystem show

```
ringsystem show ehandle label propertylist ?filterset? ?parameterdict?
y.show(property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **`ringsystem get`** command. The difference between **`ringsystem`** get **`and`** **`ringsystem show`** is that the latter does not attempt computation of property data, but raises an error if the data is not present and valid. For data already present, **`ringsystem get`** and **`ringsystem show`** are equivalent.

ringsystem sigmas

```
ringsystem sigmas ehandle label ?filterset? ?filtermode?
y.sigmas(?filters=?,?mode=?)
```

Standard cross-referencing command to obtain the labels of the σ systems the ring system overlaps with. This is explained in more detail in the section about object cross-references.

Examples:

```
ringsystem sigmas $ehandle 1
```

σ systems are a rather exotic feature and not commonly used. These are essentially descriptions of bonding interactions which use s orbitals, such as normal, covalent single bonds, or the central bond in multiple bonds. A simple double bond is described with one σ system and one π system in this representation.

ringsystem sqldget

```
ringsystem sqldget ehandle label propertylist ?filterset? ?parameterdict?
y.sqldget(property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **`ringsystem get`** command. The differences between **`ringsystem get`** and **`ringsystem sqldget`** are that the latter does not attempt computation of property data, but initializes the property value to the default and returns that default, if the data is not present and valid; and that the **SQL** command variant formats the data as **SQL** values rather than for **TCL** or **PYTHON** script processing.

ringsystem sqlget

```
ringsystem sqlget ehandle label propertylist ?filterset? ?parameterdict?
y.sqlget(property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **`ringsystem get`** command. The difference between **`ringsystem get`** and **`ringsystem sqlget`** is that the **SQL** command variant formats the data as **SQL** values rather than for **TCL** or **PYTHON** script processing.

ringsystem sqlnew

```
ringsystem sqlnew ehandle label propertylist ?filterset? ?parameterdict?
y.sqlnew(property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **`ringsystem get`** command. The differences between **`ringsystem get`** and **`ringsystem sqlnew`** are that the latter forces re-computation of the property data, and that the **SQL** command variant formats the data as **SQL** values rather than for **TCL** or **PYTHON** script processing.

ringsystem sqlshow

```
ringsystem sqlshow ehandle label propertylist ?filterset? ?parameterdict?
y.sqlshow(property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **`ringsystem get`** command. The differences between **`ringsystem get`** and **`ringsystem sqlshow`** are that the latter does not attempt computation of property data, but raises an error if the data is not present and valid, and that the **SQL** command variant formats the data as **SQL** values rather than for **TCL** or **PYTHON** script processing.

ringsystem subcommands

```
ringsystem subcommands
dir(Ringsystem)
```

Lists all subcommands of the **`ringsystem`** command. Note that this command does not require an ensemble handle, or a label.

ringsystem surfaces

```
ringsystem surfaces ehandle label ?filterset? ?filtermode?
y.surfaces(?filters=?,?mode=?)
```

Standard cross-referencing command to obtain the labels of surface patches the ring system is associated with. This is explained in more detail in the section about object cross-references.

Example:

```
ringsystem surfaces $ehandle $label
```

Note that surface patches do not need to be associated with an atom, and if they are not, they are implicitly not associated with any ring system.

ringsystem xbonds

```
ringsystem xbonds ehandle label ?filterset? ?filtermode?
y.xbonds(?filters=?,?mode=?)
```

Get labels or references of crossing bonds which are not contained in the ring system (defined as all bonds between the registered atoms in the system), but have one atom in the ring system.

The *sigma* command

The **sigma** command is the generic command used to manipulate sigma systems. The syntax of this command follows the standard schema of *command/subcommand/majorhandle/minorlabel*.

Sigma properties begin with an S_,

Pseudo sigma system labels *first*, *last* and *random* are special values, which select the first sigma system in the sigma system list, the last, or a random item.

This is the list of officially supported subcommands:

sigma append

```
sigma append ehandle label ?property value?...
s.append({?property:value,?...})
s.append(?property,value,?...)
```

Standard data manipulation command for appending property data. It is explained in more detail in the section about setting property data.

sigma atoms

```
sigma atoms ehandle label ?filterset? ?filtermode?
s.atoms(?filters=?,?mode=?)
```

Standard cross-referencing command to obtain the labels or references of the atoms in the σ system. This is explained in more detail in the section about object cross-references.

sigma bonds

```
sigma bonds ehandle label ?filterset? ?filtermode?
s.bonds(?filters=?,?mode=?)
```

Standard cross-referencing command to obtain the labels or references of the bonds between the atoms associated with a σ system. This is explained in more detail in the section about object cross-references.

sigma create

```
sigma create ehandle ?atom/atomlist?...
Sigma(eref,?aref/arefsequence/alabel?,...)
Sigma(aref,...)
Sigma.Create(eref,?aref/arefsequence/alabel?,...)
Sigma.Create(aref,...)
```

Define a new σ system from an atom set, which may be empty. A new system is always created, even if one with the same atoms already exists. Before the command is executed, the default π system set is automatically instantiated if it was not yet computed. Adding a new system invalidates properties which are sensitive to σ set changes.

The command returns the label or reference of the new σ system.

sigma defined

```
sigma defined ehandle label property
s.defined(property)
```

This command checks whether a property is defined for the σ system. This is explained in more detail in the section about property validity checking. Note that this is *not* a check for the presence of property data! The **`ens valid`** command is used for this purpose.

sigma delete

```
sigma delete ehandle ?label?...
sigma delete ehandle all
s.delete()
Sigma.Delete(eref,?sref/slabel/srefsequence?,...)
Sigma.Delete(sref,...)
Sigma.Delete(eref,"all")
```

This command deletes specific or all σ systems from the ensemble. A *sigma* property invalidation event is generated and thus the command may indirectly change the ensemble data.

The command returns the number of deleted items.

sigma dget

```
sigma dget ehandle label propertylist ?filterset? ?parameterdict?
s.dget(property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **`sigma get`** command. The difference between **`sigma get`** and **`sigma dget`** is that the latter does not attempt computation of property data, but rather initializes the property values to the default and returns that default if the data is not yet available. For data already present, **`sigma get`** and **`sigma dget`** are equivalent.

sigma ens

```
s.ens()
```

Python-only method to get the ensemble reference from a σ reference.

sigma exists

```
sigma exists ehandle label ?filterlist?
s.exists(?filters=?)
Sigma.Exists(eref,label,?filters=?)
```

Check whether this σ system exists. Optionally, a filter list can be supplied to check for the presence of specific features. The command returns 0 if the system does not exist, or fails the filter, and 1 in case of successful testing.

sigma expr

```
sigma expr ehandle label expression
s.expr(expression)
```

Compute a standard **SQL**-style property expression for the σ system. This is explained in detail in the chapter on property expressions.

sigma fill

```
sigma fill ehandle label ?property value?...
s.fill({property:value,...})
s.fill(?property,value?,...)
```

Standard data manipulation command for setting data, ignoring possible mismatches between the lengths of the lists of objects associated with the property and the value list. It is explained in more detail in the section about setting property data.

The command returns the first fill value.

sigma filter

```
sigma filter ehandle label filterlist
s.filter(filters)
```

Check whether a σ system passes a filter list. The return value is boolean 1 for success and 0 for failure.

sigma get

```
sigma get ehandle label propertylist ?filterset? ?parameterdict?
s.get(property=,?filters=?,?parameters=?)
s[property]
s.property
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

sigma groups

```
sigma groups ehandle label ?filterset? ?filtermode?
s.groups(?filters=?,?groups=?)
```

Standard cross-referencing command to obtain the labels or references of the groups the σ system overlaps with. This is explained in more detail in the section about object cross-references. An overlap between a σ system and a group is established when there are common atoms which are contained in both objects.

Example:

```
sigma groups $ehandle 1
```

sigma index

```
sigma index ehandle label
s.index()
```

Get the index of the σ system. The index is the position in the σ set of the ensemble. The first position is index 0.

Example:

```
sigma index $ehandle 99
```

sigma jget

```
sigma jget ehandle label propertylist ?filterset? ?parameterdict?
s.jget(property=,?filters=?,?parameters=?)
```

This is a variant of **`sigma get`** which returns the result data as a **JSON** formatted string instead of **TCL** or **PYTHON** interpreter objects.

sigma jnew

```
sigma jnew ehandle label propertylist ?filterset? ?parameterdict?
s.jnew(property=,?filters=?,?parameters=?)
```

This is a variant of **`sigma new`** which returns the result data as a **JSON** formatted string instead of **TCL** or **PYTHON** interpreter objects.

sigma jshow

```
sigma jshow ehandle label propertylist ?filterset? ?parameterdict?
s.jshow(property=,?filters=?,?parameters=?)
```

This is a variant of **`sigma show`** which returns the result data as a **JSON** formatted string instead of **TCL** or **PYTHON** interpreter objects.

sigma local

```
sigma local ehandle label propertylist ?filterset? ?parameterdict?
s.local(property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading and recalculating object data. It is explained in more detail in the section about retrieving property data.

sigma mol

```
sigma mol ehandle label ?filterset? ?filtermode?
s.mol(?filters=?,?mode=?)
```

Standard cross-referencing command to obtain the label or reference of the molecule a σ system is part of. This is explained in more detail in the section about object cross-references.

sigma new

```
sigma new ehandle label propertylist ?filterset? ?parameterdict?
s.new(property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

The difference between **`sigma get`** and **`sigma new`** is that the latter forces the re-computation of the property data, regardless whether it is present and valid, or not.

sigma nget

```
sigma nget ehandle label propertylist ?filterset? ?parameterdict?
s.nget(property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **`sigma get`** command. The difference between **`sigma get`** and **`sigma nget`** is that the latter always returns numeric data, even if symbolic names for the values are available.

sigma pis

```
sigma pis ehandle label ?filterset? ?filtermode?
s.pis(?filter=?,?mode=?)
```

Standard cross-referencing command to obtain the labels or references of the π systems the σ system overlaps with. This is explained in more detail in the section about object cross-references.

Examples:

```
sigma pis $ehandle 1
```

sigma ref

```
Sigma.Ref(eref,identifier)
```

PYTHON only method to get a σ reference. See **`sigma sigma`** command.

sigma rings

```
sigma rings ehandle label ?filterset? ?filtermode?
s.rings(?filters=?,?mode=?)
```

Standard cross-referencing command to obtain the labels or references of the rings the σ system is associated with. This is explained in more detail in the section about object cross-references. Rings which only partially overlap with the sigma system are included.

sigma ringsystems

```
sigma ringsystems ehandle label ?filterset? ?filtermode?
s.ringsystems(?filters=?,?mode=?)
```

Standard cross-referencing command to obtain the labels or references of the ring systems the σ system overlaps with. This is explained in more detail in the section about object cross-references.

sigma set

```
sigma set ehandle label ?property value?...
s.set(?property,value?,...)
s.set({property:value,...})
s.property = value
s[property] = value
```

Standard data manipulation command. It is explained in more detail in the section about setting property data.

sigma show

```
sigma show ehandle label propertylist ?filterset? ?parameterdict?
s.show(property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **`sigma get`** command. The difference between **`sigma get`** and **`sigma show`** is that the latter does not attempt computation of property data, but raises an error if the data is not present and valid. For data already present, **`sigma get`** and **`sigma show`** are equivalent.

sigma sigma

```
sigma sigma ehandle label
Sigma.Ref(eref,identifier)
```

Standard cross-referencing command to obtain the label or reference of the σ system as stored in property `S_LABEL`. This is explained in more detail in the section about object cross-references.

Example:

```
sigma sigma $ehandle #0
```

returns the label of the first σ system of the ensemble σ set.

sigma sqldget

```
sigma sqldget ehandle label propertylist ?filterset? ?parameterdict?
s.sqldget(property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **`sigma get`** command. The differences between **`sigma get`** and **`sigma sqldget`** are that the latter does not attempt computation of property data, but initializes the property value to the default and returns that default, if the data is not present and valid; and that the **SQL** command variant formats the data as **SQL** values rather than for **Tcl** or **Python** script processing.

sigma sqlget

```
sigma sqlget ehandle label propertylist ?filterset? ?parameterdict?
s.sqlget(property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **`sigma get`** command. The difference between **`sigma get`** and **`sigma sqlget`** is that the **SQL** command variant formats the data as **SQL** values rather than for **Tcl** or **Python** script processing.

sigma sqlnew

```
sigma sqlnew ehandle label propertylist ?filterset? ?parameterdict?
s.sqlnew(property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **`sigma get`** command. The differences between **`sigma get`** and **`sigma sqlnew`** are that the latter forces re-computation of the property data, and that the **SQL** command variant formats the data as **SQL** values rather than for **Tcl** or **Python** script processing.

sigma sqlshow

```
sigma sqlshow ehandle label propertylist ?filterset? ?parameterdict?
s.sqlshow(property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **`sigma get`** command. The differences between **`sigma get`** and **`sigma sqlshow`** are that the latter does not attempt computation of property data, but raises an error if the data is not present and valid, and that the **SQL** command variant formats the data as **SQL** values rather than for **Tcl** or **Python** script processing.

sigma subcommands

```
sigma subcommands
dir(Sigma)
```

Lists all subcommands of the **`sigma`** command. Note that this command does not require an ensemble handle, nor a sigma label.

sigma surfaces

```
sigma surfaces ehandle label ?filterset? ?filtermode?
s.surfaces(?filters=?,?mode=?)
```

Standard cross-referencing command to obtain the labels of the surface patches the σ system is associated with. This is explained in more detail in the section about object cross-references.

sigma xbonds

```
sigma xbonds ehandle label ?filterset? ?filtermode?
s.xbonds(?filters=?,?mode=?)
```

Get labels or references of crossing bonds which are not contained in the σ system (defined as all bonds between the registered atoms in the system), but have one atom in the σ system.

The *surface* command

The **`surface`** command is the generic command used to manipulate surface patches. The syntax of this command follows the standard schema of *command/subcommand/majorhandle/minorlabel*.

Surface properties begin with an O_, not S_ (which is reserved for sigma systems). The mnemonic behind this is that these are *Oberfläche* properties (German for *surface*).

Pseudo surface system labels *first*, *last* and *random* are special values, which select the first surface patch in the surface patch list, the last, or a random patch.

Examples:

```
surface get $ehandle 1 O_COLOR
```

This is the list of officially supported subcommands:

surface append

```
surface append ehandle label ?property value?...
o.append({?property:value,?...})
o.append(?property,value,?...)
```

Standard data manipulation command for appending property data. It is explained in more detail in the section about setting property data.

Example:

```
surface append $ehandle 1 O_ID "_accessible"
```

surface atoms

```
surface atoms ehandle label ?filterset? ?filtermode?
o.atoms(?filters=?,?mode=?)
```

Standard cross-referencing command to obtain the labels or references of the atoms in the surface patch. This is explained in more detail in the section about object cross-references. Note that patches may not be associated with any atom.

Example:

```
surface atoms $ehandle 1 carbon
```

returns the labels of the carbon atoms associated with the patch.

surface bonds

```
surface bonds ehandle label ?filterset? ?filtermode?
o.bonds(?filters=?,?mode=?)
```

Standard cross-referencing command to obtain the labels or references of the bonds between the atoms associated with a patch. This is explained in more detail in the section about object cross-references. In many cases, patches are associated with one or no atom. In that case, there are no bonds to retrieve.

Examples:

```
surface bonds $ehandle 1
surface bonds $ehandle 1 {1 doublebond triplebond} count
```

The first example returns all labels of the bonds between the atoms in patch one. The second example returns the number of double or triple bonds between the atoms in the patch.

surface create

```
surface create ehandle ?atom/atomlist?...
Surface(eref,?aref/arefsequence/alabel?,...)
Surface(aref,...)
Surface.Create(eref,?aref/arefsequence/alabel?,...)
Surface.Create(aref,...)
```

Define a new surface patch from an atom set, which may be empty. A new patch is always created, even if one with the same atoms already exists. Adding a new patch invalidates properties which are sensitive to patch set changes.

The command returns the label or reference of the new patch.

surface defined

```
surface defined ehandle label property
s.defined(property)
```

This command checks whether a property is defined for the surface patch. This is explained in more detail in the section about property validity checking. Note that this is *not* a check for the presence of property data! The **`ens valid`** command is used for this purpose.

surface delete

```
surface delete ehandle ?label?...
surface delete ehandle all
o.delete()
Surface.Delete(eref,?oref/olabel/orefsequence?,...)
Surface.Delete(oref,...)
Surface.Delete(eref,"all")
```

This command removes surface patches from the ensemble patch set. A *surface* property invalidation event is generated and thus the command may indirectly change the ensemble data.

The command returns the number of deleted items.

surface dget

```
surface dget ehandle label propertylist ?filterset? ?parameterdict?
o.dget(property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **`surface get`** command. The difference between **`surface get`** and **`surface dget`** is that the latter does not attempt computation of property data, but rather initializes the property values to the default and returns that default if the data is not yet available. For data already present, **`surface get`** and **`surface dget`** are equivalent.

surface ens

```
o.ens()
```

Python-only method to get the ensemble reference from a surface reference.

surface exists

```
surface exists ehandle label ?filterlist?
o.exists(?filters=?)
Surface.Exists(eref,label,?filters=?)
```

Check whether this patch exists. Optionally, a filter list can be supplied to check for the presence of specific features. The command returns boolean 0 if the patch does not exist, or fails the filter, and 1 in case of successful testing.

Example:

```
surface exists $ehandle 99
```

surface expr

```
surface expr ehandle label expression
o.expr(expression)
```

Compute a standard **SQL**-style property expression for the surface patch. This is explained in detail in the chapter on property expressions.

surface fill

```
surface fill ehandle label ?property value?...
o.fill({property:value,...})
o.fill(?property,value?,...)
```

Standard data manipulation command for setting data, ignoring possible mismatches between the lengths of the lists of objects associated with the property and the value list. It is explained in more detail in the section about setting property data.

The command returns the first fill value.

surface filter

```
surface filter ehandle label filterlist
s.filter(filters)
```

Check whether a surface patch passes a filter list. The return value is boolean 1 for success and 0 for failure.

Example:

```
surface filter $ehandle 1 [list carbon doublebond]
```

checks whether patch is associated with one or more carbon atoms and one or more double bonds. The double bond does not need to contain a carbon atom.

surface get

```
surface get ehandle label propertylist ?filterset? ?parameterdict?
o.get(property=,?filters=?,?parameters=?)
o[property]
o.property
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

surface groups

```
surface groups ehandle label ?filterset? ?filtermode?
o.groups(?filters=?,?mode=?)
```

Standard cross-referencing command to obtain the labels or references of the groups the surface patch overlaps with. This is explained in more detail in the section about object cross-references. An overlap between a surface patch and a group is established when there are common atoms which are contained in both objects.

Example:

```
surface groups $ehandle 1
```

surface index

```
surface index ehandle label
o.index()
```

Get the index of the surface patch. The index is the position in the patch set of the ensemble. The first position is index 0.

Example:

```
surface index $ehandle 99
```

surface jget

```
surface jget ehandle label propertylist ?filterset? ?parameterdict?
o.jget(property=,?filters=?,?parameters=?)
```

This is a variant of **`surface get`** which returns the result data as a **JSON** formatted string instead of **Tcl** or **Python** interpreter objects.

surface jnew

```
surface jnew ehandle label propertylist ?filterset? ?parameterdict?
o.jnew(property=,?filters=?,?parameters=?)
```

This is a variant of **`surface new`** which returns the result data as a **JSON** formatted string instead of **Tcl** or **Python** interpreter objects.

surface jshow

```
surface jshow ehandle label propertylist ?filterset? ?parameterdict?
o.jshow(property=,?filters=?,?parameters=?)
```

This is a variant of **`surface show`** which returns the result data as a **JSON** formatted string instead of **Tcl** or **Python** interpreter objects.

surface local

```
surface local ehandle label propertylist ?filterset? ?parameterdict?
o.local(property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading and recalculating object data. It is explained in more detail in the section about retrieving property data.

surface mols

```
surface mols ehandle label ?filterset? ?filtermode?
o.mols(?filters=?,?mode=?)
```

Standard cross-referencing command to obtain the labels of the molecules a patch is contained in. This is explained in more detail in the section about object cross-references. It is possible to have patches which span more than one molecule. Patches which are not associated with any atom also have no molecule association.

surface new

```
surface new ehandle label propertylist ?filterset? ?parameterdict?
o.new(property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

The difference between **`surface get`** and **`surface new`** is that the latter forces the re-computation of the property data, regardless whether it is present and valid, or not.

surface nget

```
surface nget ehandle label propertylist ?filterset? ?parameterdict?
o.nget(property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **`surface get`** command. The difference between **`surface get`** and **`surface nget`** is that the latter always returns numeric data, even if symbolic names for the values are available.

surface pis

```
surface pis ehandle label ?filterset? ?filtermode?
o.pis(?filters=?,?mode=?)
```

Standard cross-referencing command to obtain the labels or references of the π systems the surface patch overlaps with. This is explained in more detail in the section about object cross-references.

Examples:

```
surface pis $ehandle 1
```

surface ref

```
Surface.Ref(eref,identifier)
```

PYTHON only method to get a surface reference. See **`surface surface`** command.

surface rings

```
surface rings ehandle label ?filterset? ?filtermode?
o.rings(?filters=?,?mode=?)
```

Standard cross-referencing command to obtain the labels of the rings or references the surface patch is associated with. This is explained in more detail in the section about object cross-references. Rings which only partially overlap with the patch are included.

Examples:

```
surface rings $ehandle 1
surface rings $ehandle 1 [list heterocycle aroring]
```

The first example returns the labels of all rings the patch overlaps with. If the patch does not overlap with any ring, an empty list is returned. Only labels of rings in the SSSR or ESSSR set are returned, even if the currently computed ring set is larger. The second example filters the rings - only heteroaromatic rings are reported.

surface ringsystems

```
surface ringsystems ehandle label ?filterset? ?filtermode?
o.ringsystems(?filters=?,?mode=?)
```

Standard cross-referencing command to obtain the labels or references of the ringsystems the surface patch overlaps with. This is explained in more detail in the section about object cross-references.

Examples:

```
surface ringsystems $ehandle 1
```

surface set

```
surface set ehandle label ?property value?...
o.set(?property,value?,...)
o.set({property:value,...})
o.property = value
o[property] = value
```

Standard data manipulation command. It is explained in more detail in the section about setting property data.

surface show

```
surface show ehandle label propertylist ?filterset? ?parameterdict?
o.show(property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **`surface get`** command. The difference between **`surface get`** and **`surface show`** is that the latter does not attempt computation of property data, but raises an error if the data is not present and valid. For data already present, **`surface get`** and **`surface show`** are equivalent.

surface sigmas

```
surface sigmas ehandle label ?filterset? ?filtermode?
o.sigmas(?filters=?,?mode=?)
```

Standard cross-referencing command to obtain the labels or references of the σ systems the surface patch overlaps with. This is explained in more detail in the section about object cross-references.

Examples:

```
surface sigmas $ehandle 1
```

σ systems are a rather exotic feature and not commonly used. These are essentially descriptions of bonding interactions which use s orbitals, such as normal, covalent single bonds, or the central bond in multiple bonds. A simple double bond is described with one σ system and one π system in this representation.

surface sqldget

```
surface sqldget ehandle label propertylist ?filterset? ?parameterdict?
o.sqldget(property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **`surface get`** command. The differences between **`surface get`** and **`surface sqldget`** are that the latter does not attempt computation of property data, but initializes the property value to the default and returns that default, if the data is not present and valid; and that the **SQL** command variant formats the data as **SQL** values rather than for **TCL** or **PYTHON** script processing.

surface sqlget

```
surface sqlget ehandle label propertylist ?filterset? ?parameterdict?
o.sqlget(property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the *s*`urface get` command. The difference between **`surface get`** **AND** **`surface sqlget`** is that the **SQL** command variant formats the data as **SQL** values rather than for **TCL** or **PYTHON** script processing.

surface sqlnew

```
surface sqlnew ehandle label propertylist ?filterset? ?parameterdict?
o.sqlnew(property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **`surface get`** command. The differences between **`surface get`** and **`surface sqlnew`** are that the latter forces re-computation of the property data, and that the **SQL** command variant formats the data as **SQL** values rather than for **TCL** or **PYTHON** script processing.

surface sqlshow

```
surface sqlshow ehandle label propertylist ?filterset? ?parameterdict?
o.sqlshow(property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **`surface get`** command. The differences between **`surface get`** and **`surface sqlshow`** are that the latter does not attempt computation of property data, but raises an error if the data is not present and valid, and that the **SQL** command variant formats the data as **SQL** values rather than for **TCL** or **PYTHON** script processing.

surface subcommands

```
surface subcommands
dir(Surface)
```

Lists all subcommands of the **`surface`** command. Note that this command does not require an ensemble handle, nor a surface label.

surface surface

```
surface surface ehandle label
Surface.Ref(eref,identifer)
```

Standard cross-referencing command to obtain the label or reference of the surface patch as stored in property `O_LABEL`. This is explained in more detail in the section about object cross-references.

Example:

```
surface surface $ehandle #0
```

returns the label of the first surface patch of the ensemble patch set.

The *table* Command

The **table** command is the generic command used to manipulate **CACTVS** table objects. The syntax of this command follows the standard schema of *command/subcommand/tablehandle/args.* Table objects are major objects just like ensembles, reactions or datasets and have, in addition to object-specific commands, the standard command set for major objects and can possess table-level properties, which start by convention with prefix `T_`. Example:

```
table get $thandle T_XYPLOT
```

Because tables are pure major objects without an internal set of minor objects, there are no minor object labels. Nevertheless, there are mechanisms to address columns, rows and cells. Commands which address rows or columns use a syntax schema of *command/subcommand/tablehandle/address/args*.

Depending on the context, column and row arguments to table commands can be either single addresses, or address ranges. Single addresses are identified either by a numerical row or column index starting with zero, the magic names *end* or *last*, or a symbolic name which is was assigned to the row or column at some time during its existence. Columns can have additional alias names. Examples:

```
table getcol $thandle last datatype
table getcol $thandle 0 data
```

Address ranges are specified by two simple addresses, separated by a dash character. These ranges can also be open, meaning that if there is no name component to the left of the dash, start column or row zero is implied, and if there is no name component to the right of the dash, the rightmost column or bottommost row is the end of the range. When a column or row range without any range dashes is specified, the range only includes that single item. Finally, the magic names *all* or * select all current columns or rows. Examples:

```
table delrow $thandle 3-last
table dupcol $thandle E_NAME end
```

Table objects can store output formatting information in different places - globally, on columns, rows and individual cells. The precedence for these formats is *cell>row>column>global.* The fonts, font sizes and colors, for which there can only be one value, that are used for the output of a specific cell are determined by the location with the highest precedence where the attribute is set to a definite value (i.e. in the case of fonts, the location where a font name and not an empty string is provided). General formatting flags are bit-ored from all locations. There is currently no method to suppress the use of a formatting flag on output which is set in any checked location. Example:

```
table setcol $thandle 0 bgcolor white
table setrow $thandle 1 bgcolor red
```

All cells in row one are output with a red background, provided that there are no cell-level overrides, and that the output format supports cell coloring.

Tables possess an internal utility dataset. Its handle can be retrieved with

```
table get $thandle dataset
```

This internal dataset is useful to store objects which are referenced by rows. Normally, these referenced objects which are introduced by command such as **table addens** or **table addreaction** are not destroyed when the table is deleted. However, when they are moved into the internal dataset, are will be deleted with the table, just like other objects in a dataset object when that dataset object is destroyed. In case of table file formats which store both table cell data and structure or reaction objects, the associated objects are moved to the internal dataset when such a file is read.

This is the list of officially supported subcommands:

table addcolumn

```
table addcolumn tablehandle columntype ?property|datatype|expression?
   ?name? ?position? ?width? ?startvalue?
t.addcolumn(property=,?name=?,?position=?,?width=?,?startvalue=?)
```

This command adds a new column to the table. All data values for the new column in existing rows are initially set to **NULL**, with the exception of function columns, which are set to the computed value as far as that is possible from the current table content, and sequences, which are initialized with a number sequence.

table addcol is a shortened alias command name.

The *startvalue* parameter is only used for sequence type columns.

The column type argument can be one of:

- *data*
  A data column with cells of a specific data type which is not associated with a **Cactvs** property, or where the property association is unknown. If no data type is supplied in the next argument, the default is *double*.
- *function*
  A data column with cells which contain dynamically computed values. The next argument is the SQL-style function expression. If that argument is not provided, the function result for the cell values will always be **NULL**. formula is an alias column type name for this type.
- *image*
  Add an structure or reaction image column. This is a special type of property column. The property behind this column is dynamically adapted according to chosen output formats. For example, it is `E_GIF` for **HTML** pages, `E_EMF_IMAGE` for Windows **MS Excel**, and `E_PICT_IMAGE` for Mac **MS Excel** output. For this column type, the next optional argument is immediately the column name, not the detailed type specification.
- *none*
  An unspecified column. Useful as place holder in case this is defined later, or to add spacer columns. For this column type, the next optional argument is immediately the column name, not the detail type specification.
- *property*
  A data column with cell values which are linked to a **Cactvs** property. The fundamental data type, as well as enumeration values, formatting conventions etc. are all inherited from the property definition. If the property name is not supplied in the next argument, the default is property `E_NAME`. Property-associated columns are especially useful for automatic transfer of chemical object data into a table by means of the **table addens** and **table addreaction** commands.
- *sequence*
  An integer data column automatically filled with sequential numbers starting with one for the first row, or another value if an explicit start value is given. For this column type, the next optional argument is immediately the column name, not the detailed type specification.

The content of the argument of the column type specification must be appropriate to the previous argument, i.e. a data type for columns of type *data*, a property name for columns of type *property*, or a parseable **SQL** expression for columns of type *function*.

For data and property type columns, the column type may also be omitted and the property or data type name written immediately. These two lines are equivalent:

```
table addcol $thandle property E_XLOGP2 xlogp
table addcol $thandle E_XLOGP2 xlogp
```

In the **Python** interface, the column type and column detail are rolled into a single argument. It can either be a single item, which is equivalent to the single-argument **Tcl** form, or a tuple with the type and detail pair corresponding to the two **Tcl** command arguments. Above example in **Python**:

```
t.addcol((“property”,”E_XLOGP2”),”xlogp”)
t.addcol(“E_XLOGP2”)
```

Columns can be given a user-defined name. If the optional name argument is not given, a synthetic name is automatically supplied. For property columns, it is the name of the property. For other columns, it is the column index appended to the column type, as in *data1* for a data column with column index one.

By default, columns are appended to the right. This is also done if the optional position argument is *end*, an empty string, or a column index beyond the current maximum column index. Otherwise, the column is inserted into the specified position and all other columns behind it are moved one position to the right. No columns are overwritten. Existing cell data is also moved if necessary. It is not possible to add a column to the right beyond the rightmost column in a way that undefined columns result. It is possible to use a column name in addition to a numerical index. In that case, the new column is added to the right of the named column.

The next optional argument defines the column width, which is only used for output formatting. It is the same as the column attribute *width* and described in the paragraph on the **`table setcol`** command. If this argument is omitted, or an empty string, the width is undefined and defaults are used. The optional start value argument is only used for sequence columns.

The return value of the command is the new number of columns in the table.

table addrow

```
table addrow tablehandle ?name|#auto? ?position? ?celldatalist? ?cellobjlist?
   ?rowobject?
t.addrow(?name=?,?position=?,?celldata=?,?cellobjects=?)
```

Add a row to an existing table. By default, all cell data values of the newly added row are set to **`NULL`**.

Table rows can be named. If the name argument is omitted, set to an empty string (or **`None`** for **Python**), or *#auto* is used as magic name, the standard automatically generated row name, of the form **Row***rowcount*, is used, with the row count replaced by the integer value. This format can be overridden by setting a table row name template format string (*autorowformat* table attribute). Note that this name uses the total row count of the table after adding the new row, not the insert position. This makes it less likely to accidentally generate duplicate row names if multiple rows are inserted into the same position.

By default, the new row is appended to bottom of the table. This also happens if the optional position argument is *end*, or larger than the current maximum row index, which starts with zero. If any other valid row index is given, the new row is inserted into that position and the rest of the rows behind it are moved. No existing rows are overwritten. It is not possible to add a row beyond the current end of the table in such a fashion that undefined rows result.

Next, it is possible to initialize the cell values of the new row. If that option is chosen, the length of the column data list must smaller then, or the same length, as the number of columns in the table, and every column data value must be decodable according to the respective column data type.

The optional row object argument can be either an ensemble or reaction handle. If it is specified and not an empty string, the object becomes associated with the new row, as by the **`table addens`** or **`table addreaction`** commands. If an explicit empty string is used, the row is explicitly not associated with an ensemble or reaction, and no attempt is made to find and decode a row object source in the column data when it is requested at a later time.

The same processing applies a a cell object list is used. The cell objects (ensembles or reactions) are transferred to the table and henceforth are under the control of the table and no longer deletable by normal means. They are only destroyed if the associated table cell is deleted. By default the new table cells have no cell object, and this also remains the case if the object list element for a cell is an empty string. Cell objects to be transferred cannot be already undeletable (e.g. a property value or the cell object of another cell). If such objects need to be set, they must be duplicated first.

If data is extracted from chemical objects and stored in a table object, it is usually more convenient to use the **`table addens`** and **`table addreaction`** commands than to script sequences of **`table addrow`** statements.

The table must be editable for this command to be usable.

The return value of the command is the new number of rows in the table.

table adddataset

```
table adddataset tablehandle datasetlist ?objclass? ?filterlist? ?position? ?mode?
t.adddataset(datasets=,?objclass=?,?filters=?,?position=?,?mode=?)
```

This command performs a **`table addens`** command for each ensemble in the datasets, and a **`table addreaction`** command for each reaction. The command arguments are interpreted as described in each specialized command, and have the same defaults if they are not set explicitly.

If a dataset contains other objects besides ensembles and reactions, these objects are ignored in the table data setting operation. If the mode is set to *destroy*, not only the ensembles or reactions in the dataset are deleted, but the complete dataset with all its content, including other objects which are not reactions or ensembles. With mode *move*, ensembles and reactions are removed from the source dataset and transferred to the internal table dataset. In this mode, other dataset content objects that are not reactions or ensembles remain in the source dataset, and the source datasets persist. Finally, mode *preserve* (the default) retain all object memberships, and again the source datasets persist.

The command returns the number of added table rows.

table addens

```
table addens tablehandle enslist ?objclass? ?filterlist? ?position? ?mode?
t.addens(ens=,?objclass=?,?filters=?,?position=?,?mode=?)
```

Capture data from a list of ensembles in a table object. One or more new rows are added, and all table columns for these rows which refer to property data present or computable on the ensemble are filled. In addition, references between the new rows and the ensemble are registered.

The object class determines how many rows are added per ensemble. It can be *ens*, or the type of any ensemble minor object. One row is added for each ensemble minor object (or the ensemble proper, in case the object class is *ens*), provided it passes the filters if a filter list is specified. If the object class argument is not specified, or given as an empty string (or **None** in **PYTHON**) or the special value *auto* or *#auto*, an attempt is made to determine it automatically. If only data columns of ensemble properties are found, it is then set to *ens*, but if there are any data columns of properties related to ensemble minor objects except molecules, the object class is that class. If there are only molecule and ensemble properties, the class is *mol*. For each object of the selected class in the ensemble, one row is added, and the cells filled with the data extracted from the associated ensemble or ensemble minor object.

For example, assume the object class is *atom*; let there be table data columns of atom, molecule, and ensemble properties; and let there be four atoms and two molecules in an ensemble to be processed. In that case, total of four rows are added, with the ensemble property data cells holding the same data for all four rows, and the molecule property data cells holding two duplicates for two rows each, and every atom data cell the data of one of the four different atoms. As long as all minor objects for which a row is added (with the exception of molecules) are completely contained in exactly one larger object of the involved classes, property types can be freely combined. For example, atom properties and molecule or ensemble properties mix, because every atom is only a member of one molecule and one ensemble. However, the results from combining atom and bond or ring properties are, while still deterministic, probably not useful for any real application. The same is true for a mismatch of the column properties and an explicit object class, i.e. combining atom properties with a bond object class is not likely to be useful.

If the filter list argument is specified, rows are only added for the objects which pass the filter. Those which do not pass the filter are silently skipped. The type of filters which are suitable are usually determined by the selected object class. For example, if a row is added for each atom, atom filters are certainly useful. However, more exotic combinations are possible - for example, atom filters may be used in combination with a *mol* object class - in that case only data rows for molecules in which one or more atoms pass the filter are added.

Only cells in property and function columns are populated with values by this command. Any cells in pure-data columns without a property descriptor are set to **NULL**.

By default, the new row or rows are added at the bottom of the table. If desired, a different row index may be given, with index zero resulting in the insertion of the first new row as the first table row, and any additional new rows immediately behind it. The special position *end* may be used to explicitly append to the table.

By default, the ensemble objects providing the data are preserved, and they remain in their original dataset membership context. The optional *mode* argument can be set to change their fate. Its possible values are *preserve* (the default), *delete* (the ensembles are destroyed after setting the table row data) and *move* (the ensembles are moved to the internal dataset object of the table). In mode *delete*, the relationship between rows and the ensemble is not preserved, because the ensemble is gone after the row addition.

The command returns the number of added table rows.

Example:

```
table addens $th $eh
```

table addmolfile

```
table addmolfile tablehandle molfilelist/filenamelist ?objclass? ?filterlist?
   ?position? ?mode?
t.addmolfile(molfiles=,?objclass=?,?filters=?,?position=?,?mode=?)
```

This command performs the equivalent of a **`table addens`** or **`table addreaction`** command for every ensemble or reaction which can be read from the argument structure file handles. The type of object read from the file, in case there are multiple possibilities, is determined by the configuration of the file handle. Please refer to the section on the **`table addens`** and **`addreaction`** commands for additional explanations. The command arguments after the file handles are interpreted as in these commands.

Input starts from the current file position of every specified file handle. The command fails if any file cannot be read to the end. The files are positioned at **`EOF`** after a successful operation.

If a file handle list argument is not a *molfile* object handle, an attempt is made to interpret it as the name of a structure or reaction data file. If such a file exists, is readable, and of a recognized file format, it is transiently opened in read-only mode with default settings and automatically closed when the command completes. This is equivalent to the handling of transient files in the **`molfile`** command.

Different from the **`table adddataset/addens/addreaction`** commands, the default object addition mode of this command is *delete*. In this mode, the read ensemble or reaction objects are deleted after the table cells have been filled from the current object, so there is no persistent table row association with a structure object. In mode *preserve*, the ensembles remain in memory, and in mode *move*, they are also kept, but additionally moved to the internal table dataset. The latter two modes can of course greatly increase the memory requirements, so these modes should not be used indiscriminately.

`table addfile` is a command alias.

The command returns the total number of added table rows.

table addreaction

```
table addreaction tablehandle reactionlist ?objclass? ?filterlist? ?position?
   ?mode?
t.addreaction(reactions=,?obvjclass=?,?filters=?,?position=?,?mode=?)
```

Capture data from one or more reactions in a table object. One or more new rows are added, and all table columns for these rows which refer to property data present or computable on the reactions are filled. In addition, a reference is created between the new table rows and the reactions, or, with an *ens* object class, the reaction ensembles.

If the object class is *reaction*, a single row is added per reaction and reaction-level property data is copied. Alternatively, it may be specified as *ens*, which is equivalent to the execution of one **`table addens`** command for every reaction ensemble. If this argument is not specified explicitly, or given as an empty string or the special value *auto* or *#auto*, it is automatically determined from the column types present. If any data columns are reaction properties, the object class is set to *reaction*, otherwise *ens*.

If the filter list argument is specified, only reactions or ensembles which pass the filter are added. Those which do not pass the filter are silently skipped.

By default, the new row or rows are added at the bottom of the table. If desired, a different row index may be specified, with index zero resulting in the insertion of the first new row as the first table row, and any additional new rows immediately behind it. The special position *end* may be used to explicitly append to the table.

By default, the ensemble objects providing the data are preserved, and they remain in their original dataset membership context. The optional *mode* argument can be set to change their fates. Its possible values are *preserve* (the default), *delete* (the reactions and all their ensembles are destroyed after setting the table row data) and *move* (the reactions are moved to the internal dataset object of the table). In mode *delete*, the relationship between rows and the reaction or reaction ensembles is not preserved, since the reaction is gone after the command.

The command returns the number of added table rows.

table addretrieval

```
table addretrieval tablehandle ?field?...
t.addretrieval(?field?,...)
```

Add columns to the table, using the syntax of the result retrieval argument of commands such as **`molfile scan`** or **`dataset scan`**. The modification of the table column set is the same as using a scan command in the to-table result output mode with an existing table as output target. Please refer to the documentation of **`molfile scan`** for details on the syntax.

Example:

```
table addretrieval $th {*}[knode param $kh customretrieval]
```

This is an example for the preparation of an output table for a **CACTVS KNIME** node in the configuration function. In the **KNIME** context, an output table must be fully configured by the configuration script, so this cannot be delegated to the **`molfile scan`** command run in the execution script of the node.

table append

```
table append tablehandle ?property value?...
t.append({?property:value,?...})
t.append(?property,value,?...)
```

Standard data manipulation command for appending property data. It is explained in more detail in the section about setting property data.

The command returns the first data value.

Example:

```
table append $ehandle T_COMMENT "\nI still do not think this data makes sense!"
```

table assign

```
table assign tablehandle srcprop dstprop
t.assign(srcproperty=,dstproperty=)
```

Assign property data to another property on the same table. Both properties must be associated with the table object class. This process is more efficient than going through a pair of **`table get/table`**

`set` commands, because in most cases no string or **TCL/PYTHON** script object representations of the property data need to be created.

Both source and destination properties may be addressed with field specifications. A data conversion path must exist between the data types of the involved properties. If any data conversion fails, the command fails. For example, it is possible to assign a string property to a numeric property - but only if all property values can be successfully converted to that numeric type. The reverse example case always succeeds, out-of-memory errors and similar global events excluded.

The original property data remains valid. The command variant **`table rename`** directly exchanges the property name without any data duplication or conversion, if that is possible. In any case, the original property data is no longer present after the execution of this command variant.

Examples

```
table assign $th T_IDENT T_NAME
```

table blockloop

```
table blockloop tablehandle column rowvariable ?maxblocks? ?offset? body
t.blockloop(column=,function=,?maxblocks=?,?offset=?,?variable=?)
```

This command is a convenience function for looping over the contents of a table. It is a more complex version of the **`table loop`** command. It works on blocks of rows which have the same value in a column instead of a single row.

In the **TCL** variant, the contents of one or more rows are stored as a nested list in the global **TCL** row variable. The inner objects are either lists that contain one cell data element per table column (with a *list* table iterator) or dictionaries with column names as keys. In each iteration, after the variable has been updated, the **TCL** code in the *body* argument is executed. The standard **TCL** loop control constructs *break* and *continue* work as expected within the loop. The *iteratorstyle* table attribute controls the formatting of the elements of the outer list (i.e. whether this are lists, or dictionaries. The default iterator mode is *list*.

The length of the nested iteration argument list is determined by the number of consecutive rows which have the same value in the data cell of the block column as the current row. The next iteration of the loop continues with the first row which has a value in the block column data cell that is different from the current value. Note that this command does not sort the table. If the same block column cell value appears in rows which are not consecutive, multiple blocks are processed with the same value. If the block column is an empty string, or the special row name *#name*, the block membership is determined by the row name.

By default, the iteration continues until the end of the table, but an upper limit may be specified in the optional *maxblocks* parameter. If this parameter is explicitly set to a negative value, the loop runs to the end of the table. The default starting point of the loop is the first row. This can be changed by giving an explicit offset in the second optional parameter.

The **PYTHON** version of the loop method does intentionally have a different argument sequence for convenience. The function argument may either be a multi-line string (similar to the **TCL** construct), or a function reference. Functions are called with the table reference and a list of current block data rows as two arguments, and have their own context frame, so that the specification of a reference variable is not generally useful in that call style, though is is allowed. For string function blocks the code is executed in the local call frame, and the variable with the current object reference is visible locally. Script code blocks must be written with an initial indentation level of zero. Within the

Python functions, the normal *break* and *continue* loop control commands cannot be used to to scope limitations. Instead, the custom exceptions *BreakLoop* and *ContinueLoop* can be raised. These are automatically caught and processed in the loop body handler code.

Example:

```
table cluster $thandle E_SCREENING_RESULT jarvispatrick {colname clusters}
table sort $thandle clusters
table blockloop $thandle clusters rowvar {
   foreach row $rowvar {
      lassign $row cpdname clusterid
      ...
   }
}
```

The loop is executed once per cluster with the variable set to the row data block of all structures in that cluster.

The return value of the command is the number of loop iterations processed. The last value of the loop variable remains accessible outside the loop.

The commands **`table dictblockloop`** and **`table listblockloop`** are variants of this command which ignore the configured iterator style attribute of the table.

table cast

```
table cast tablehandle dataset/ens/reaction/table ?propertylist?
t.cast(objectclass=,?properties=?)
```

Transform the table into a different object. With the exception of the *table* target object class, which does nothing, the table is destroyed in the process. Depending on the target object class, the result is as follows:

- *dataset*
  A new dataset which contains all the reaction and ensemble objects associated with the table rows, without duplicates. Rows without an ensemble or reaction association generate a new empty ensemble. The property set of the objects in the new dataset is automatically augmented with the data of the associated table row.
- *ens*
  The ensemble associated with the first table row, or an empty ensemble if no such association exists. The ensemble property set is automatically augmented with the row data.
- *reaction*
  The reaction associated with the first table row, or an empty reaction if no such association exists. The reaction property set is automatically augmented with the row data.
- *table*
  Only supported for the sake of completeness, this mode does nothing.

If the optional property list is specified, an attempt is made to compute the listed properties before the cast operation, so that they may become a part of the new object. No error is raised if a computation fails.

The command returns the handle or reference of the new object, or the original input object in case of mode *table*.

table celldata

```
table celldata tablehandle row column ?value? ?flags?
t.celldata(row=,column=,?value=?,?flags=?)
```

In the simple form without the optional flags argument, this command is essentially a shortcut for the **`table setcell`** and **`table getcell`** commands with the *value* attribute.

If the flags argument is used, setting of the cell data can be modified. The flags argument can be one or more of the following words:

- *append*
  The data is appended to the current cell data, if the data type of the cell supports the concept of appending. The default is to replace it.
- *clear*
  Set the cell to a **`NULL`** value. The value argument is ignored.
- *force*
  Override any editing locks on the table, column, row or cell.
- *recall*
  Return the old value of the cell before the update as command result.
- *recallnew*
  Return the new value of the cell after the update. Because of the formatting implied by, for example, columns which hold property values and where the property definition contains enumerations, constraints, precision limits, etc., this may not be exactly the value argument which is input.
- *settime*
  Set the update time stamp on the table. By default, this is not done due to its inherently expensive character because it needs to issue a system call.

table clear

```
table clear tablehandle
t.clear()
```

Reset a table. All rows and columns as well as table-level property data are removed, and user-configurable attributes are reset to default values. However, the table handle remains valid and can be re-used to set up a new table.

The table needs to be editable to allow this command to succeed.

The command returns the table handle.

table clone

```
table clone srctablehandle ?dsttablehandle? ?columnrangelist?
t.clone(?target=?,?columnranges=?)
```

Copy the table definition from the source to another table. If a destination table is specified, all cells, rows and columns of the destination table are deleted before the information is copied from the source.

If the destination table handle or reference is omitted, or specified as an empty string (or **`None`** for **PYTHON**), or the special names *new* or *#auto* are used, a new table is created and the command has a similar effect as **`table dup`**.

The optional column range list argument can be used to copy only some of the columns. By default, all columns are copied.

This command is similar to **`table copy`**, except that no row and cell data is copied. The destination table has the same column layout and other global attributes of the source table, but now rows or cells.

The command returns the handle or reference of the destination table, which may just have been newly created.

table clonecolumns

```
table clonecolumns srctablehandle columnrangelist dstablehandle ?position?
t.clonecols(?columnranges=?,?target=?,?position=?)
```

Transfer the definitions of columns in the source table to the destination table.

If the destination table handle or reference is omitted, or specified as an empty string (or **`None`** for **Python**), or the special names *new* or *#auto* are used, a new table is created and the command has a similar effect as **`table dup`**.

Different from the **`table clone`** command with a column range list argument, the destination table is not reset when this command is run. The transferred column definitions are added to the existing set. In case of column name collisions, the names of the added columns are automatically modified. If the destination table already contained rows, the cells of the new columns are all **`NULL`** values. No cell data is copied by this command.

If no position is set, the new columns are appended to the right. Otherwise, they are inserted at the specified position, or, if the position is defined by a column name, to the right of this column.

The short form **`table clonecols`** is an alias to this command.

The command returns the handle or reference of the destination table.

table cluster

```
table cluster tablehandle columnrangelist ?method? ?parameterdict?
t.cluster(columnranges=,?method=?,?parameters=?)
```

Perform clustering on table data and add the clustering results as an additional table column. The currently supported methods are *kmeans, fuzzykmeans, centroid, ward* and *jarvispatrick*. *kmeans* is the default method. The optional parameter dictionary argument is a standard keyword/value dictionary. Currently the following parameters are recognized:

- *colname*
  The name of the newly added result column. If it is not set, or set to an empty string, the default name is *cluster*.

- *epsilon*
  The epsilon value, i.e. the minimum total value the cluster centroids need to have moved as result of the last iteration to start another iteration cycle. Only used in fuzzy **KMeans** clustering, where the default value is 0.001. If the total movement of the centroids is less than that, the iteration is stopped immediately.

- *exponent*
  The exponent used in the distance law to compute the location of the cluster centroids. Only used in fuzzy **KMeans** clustering, where the default is 1.5.

- *maxncycles*
  The maximum number of iteration cycles. Only used in normal and fuzzy **KMeans** clustering. If the cluster membership does not change during a cycle in normal **KMeans** clustering, or the total movements of the cluster centroids is less than the epsilon value in fuzzy **KMeans** clustering, iteration is stopped early even if the maximum number of iterations has not yet been reached.

- *ncluster*
  The number of clusters to find. Used in normal and fuzzy **KMeans** clustering and for the **Centroid** and **Ward** methods. **Jarvis-Patrick** clustering determines the number of clusters algorithmically.

- *ncommon*
  The minimum number of elements needed to be in common among the number of examined closest neighbor points for joint cluster membership. Only used in **Jarvis-Patrick** clustering. This value must be less than or equal to the *nexamine* parameter. The default value is one.

- *nexamine*
  The number of closest neighbor points to examine for being in the joint neighborhood of two points which could potentially be in the same cluster. Only used in **Jarvis-Patrick** clustering. The default value is two.

- *nsteps*
  The number of merge steps. Only used in the **Centroid** and **Ward** methods. The default value is the minimum of 5 and the number of eligible rows.

- *position*
  The column position where the result column is inserted. It can be either a numerical index, or the special name *end* for addition to the right of the current columns. which is the default.

The data type of the result column depends on the selected clustering method:

- *centroid*
  The result column is an integer vector with a length equal to the number of merge steps plus one. Initially, and this is recorded in vector element zero, every eligible row is in its own cluster, so all cluster IDs in the range from one to *nrows* are used. After each merge step, a new vector index is filled for all rows. All values are the same as that of the previous index, except that the cluster ID of the cluster with the higher value of the merged pair is withdrawn and its value replaced by the ID of the other cluster in all rows in the absorbed cluster. If a full merge is performed, all rows will end up as members of cluster one. Rows which are not eligible for clustering retain a cluster number of zero during all merge steps. The merge distances for all steps are stored in a double vector as column header data.

- *fuzzykmeans*
  The result column is a double vector, with the number of elements equal to the requested cluster number (*ncluster* parameter). Every element holds the fractional membership value for that cluster.

- *jarvispatrick*
  The result column is of type simple integer. It holds the number of the cluster the row is a member of. Cluster numbers begin with one. In case the row data is **`NULL`** or otherwise excluded from clustering, it is set to zero.
- *kmeans*
  The result column is of type simple integer. It holds the number of the cluster the row is a member of. Cluster numbers begin with one. In case the row data is **`NULL`** or otherwise excluded from clustering, it is set to zero.
- *ward*
  The result data format is the same as for the *centroid* method.

Data columns which are used as input data must be convertible into floating-point values. Multiple columns may be used in parallel, but it is the responsibility of the script writer to perform any scaling and other data preparation steps.

The return value is the number of clusters found, after the last iteration or merge step, if applicable.

table compare

```
table compare tablehandle1 tablehandle2 ?rowmode? ?comparisoncolumn?
t.compare(table2=,?rowmode=?,?comparisoncolumn=?)
```

This command is a dry-run version of the **`table merge`** command. Instead of actually modifying the first table, this command sets the *selected* row attribute on both tables to indicate which rows from both tables would be present in a new combined table.

The meaning of the parameters are the same as in the **`table merge`** command.

table copy

```
table copy srctablehandle ?dsttablehandle? ?columnrangelist? ?rowmode?
t.copy(?target=?,?columnranges=?,?rowmode=?)
```

Copy the table definition and table cell data to another table. If a destination table is specified, all cells, rows and columns of the destination table are deleted before the information is copied from the source.

If the destination handle is omitted, or specified as an empty string (or **`None`** for **PYTHON**), or the special names *new* or *#auto* are used, a new table is created and the command has a similar effect as **`table dup`**.

The optional column range list argument can be used to copy only some of the columns. By default, all columns are copied.

The default row mode is *all*, where all source table rows are copied. Other supported modes are *selected* (only rows for which the selection flag is set are copied) and *unselected* (only rows for which the selection flag is not set are copied).

A related command is **`table clone`**, which also adjusts the column definitions and other attributes of the destination table to match that of the source, but does not transfer row and cell data.

The command returns the handle or reference of the destination table, which may just have been newly created.

table copycolumns

```
table copycolumns srctablehandle columnrangelist dstablehandle ?position?
t.copycolumns(columnranges=,target=,?position=?)
```

Transfer the definitions of columns in the source table to the destination table, and also copy the cell content.

The copied column definitions are added to the destination table, which is not reset. In case of column name collisions, the names of the added columns are automatically modified. The number of rows in the destination table is not changed. Existing cell data from the source table is copied to the same row in the destination table. If there are more rows in the source table, the extra source cells are ignored. If there are more rows in the destination table, the extra destination cells have **`NULL`** values.

If no position is set, the new columns are appended to the right. Otherwise, they are inserted at the specified position, or, if the position is defined by a column name, to the right of this column.

`table copycols` is a shortened alias to this command.

The command returns the handle or reference of the destination table.

table copyrows

```
table copyrows srctablehandle rowrange dsttablehandle ?position?
t.copyrows(rowrange=,target=,?position=?)
```

Copy a range of rows from one table to another. If no explicit position is given, the new rows are appended to the destination table. The source table retains the copied rows. It is allowed to use this command to duplicate table rows within a single table. The destination table is not required to have the same column layout, just the same column names for matching columns. Columns not present in the destination table are added (with **`NULL`** data for existing rows), and the cell order is adapted while copying if necessary. If a pair of columns has no compatible datatype, an attempt at data conversion to the destination column type is made. If that fails, the destination cell is silently set to **`NULL`**.

The command returns the handle or reference of the source table.

table create

```
table create packstring|aid
table create ?property|image|none|enshandle|reactionhandle|datasethandle|
   molfilehandle|tablehandle?...
Table(packstring/aid)
Table.Create(packstring/aid)
Table(?property/image/none/eref/xref/dref/fref/tref?,...)
Table.Create(?property/image/none/eref/xref/dref/fref/tref?,...)
```

Create a new table. The return value of the command is the new table handle or reference.

The first variant of the command generates a fully initialized table with row and column specifications, cell data and potentially table properties. The single argument can either be a serialized packed table string (see **`table pack`** command), or a **PubChem** assay identifier (**AID**). **AID**s are simple integers, or integers prefixed with *AID* or *TID*. For these, the full assay content is downloaded from **PubChem**. Depending on the size of the assay, this can take a minute or more. The magic table name *pse* creates a table with atom information from the periodic system of elements.

Example:

```
set th [table create 67]
```

The second variant initially creates an empty table. By supplying additional arguments, one or more columns can be specified in a single statement, and/or ensemble and reaction data added to the specified columns. These are shorthand notations for **`table addcol`** or **`table addens/addreaction/adddataset/addfile`** commands. Optional arguments that are a property name, or the special names *image* or *none* are equivalent to

```
table addcol $thandle $arg
```

(one such statement per additional argument) and the chemistry object handle arguments are handled the same way as writing one or more of

```
table addens $thandle $arg
```

or

```
table addreaction $thandle $arg
```

If a table handle is used as an argument, its column structure and global table properties and attributes, but not its rows and cell data, are copied as with the **`table clone`** command.

The **Python** version accepts both names or handles and references for properties or data source objects.

The full command equivalents of the shortcuts offer more options for control of the row or column addition modes, so this abbreviated command variant can only be only used in simple cases. Ensemble or reaction handles as row data sources should be supplied after column type or table handle arguments, otherwise the cell data of these columns rows already existing when a column is added is set to **`NULL`**.

Example:

```
set th [table create E_NAME E_SMILES E_XLOGP2 E_WEIGHT image $eh1 $eh2]
```

table data

```
table data tablehandle ?rowrangelist? ?columnrangelist? ?nullvalue?
t.data(?rowrange=?,?columnrange=?,?nullvalue=?)
```

Extract cell data from a table as a nested list. By default, the full table content is returned. The optional parameters allow the selection of a specific subset of the rows and/or columns. The final optional parameter can be used to control the output format of **`Null`** data. If that parameter is omitted, the global style defined by the *undefined* table attribute (see **`table get/set`**) is used.

For historical reasons, the variant **`table print`** is an alternative name for this command.

Example:

```
set elements [lsort [table data table0 all symbol]]
```

This command retrieves an alphabetically sorted list of the atom symbols of the **PSE**.

table dataset

```
table dataset tablehandle ?filterlist?
t.dataset(?filters=?)
```

If the table is a member of a dataset, report the handle or reference of the dataset object. If the table is not a member of a dataset, or does not pass all of the optional filters, an empty string (**`None`** for **PYTHON**) is the result.

This command is different from **`table get $`*`thandle dataset`***. The latter retrieves the handle of the internal dataset which is an integral part of the table data structure.

table defined

```
table defined tablehandle property
t.defined(property)
```

This command checks whether a property is defined for the table. This is explained in more detail in the section about property validity checking. Note that this is *not* a check for the presence of property data! The **`table valid`** command is used for this purpose.

table delcolumns

```
table delcolumns tablehandle ?columnrange?..
t.delcolumns(?columnrange?,...)
```

Delete a set of column ranges, including cell data under these columns, from the table. All selected columns are deleted in a single operation, so all column numbers used in the arguments refer to the original table, not those after deletion of column sets defined in arguments to the left in the same command.

The command may also be written as **`table delcols`** or **`table delcol`**.

The return value is the number of deleted columns.

table delete

```
table delete ?tablehandle?...
table delete all
t.delete()
Table.Delete("all")
Table.Delete(?tref/trefsequence/thandle?,...)
```

Destroy one or more table objects. The special handle *all* can be used to remove all deletable tables. Tables with the *undeletable* status flag (see **`table set`**) are not affected. It is also not possible to delete the three system tables (element data, expansion fragments and SMILES macros). Finally, tables with a reference count of two or more, as they are produced by the definition of table slices, are also not deleted by this command. The referring slice tables need to be removed first before the underlying base table can be deleted.

Objects referenced by table rows, as introduced by **`table addens`** or **`table addreaction`** commands, are usually not deleted, except when they were put into the internal table dataset object.

The return value is the number of successfully deleted tables.

table delrows

```
table delrows tablehandle ?rowrange?..
t.delrows(?rowrange?,...)
```

Delete a set of row ranges, including cell data in these rows, from the table. All selected rows are deleted in a single operation, so all row numbers used in the argument refer to the original table, not those after deletion of rows sets defined in arguments to the left in the same command.

The *all* range is internally optimized to delete all current rows in one step, and to ignore all other specifications.

The return value is the number of deleted rows.

Example:

```
table delrows $thandle all
```

table dget

```
table dget tablehandle propertylist ?filterset? ?parameterdict?
t.dget(property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **`table get`** command. The difference between **`table get`** and **`table dget`** is that the latter does not attempt computation of property data, but rather initializes the property values to the default and return that default if the data is not yet available. For data already present, **`table get`** and **`table dget`** are equivalent.

table dictblockloop

```
table dictblockloop tablehandle column rowvariable ?maxrows? ?offset? body
t.dictblockloop(column=,function=,?maxblocks=?,?offset=?,?variable=?)
```

This command is a variant of the **`table blockloop`** command. It stores the row data as a dictionary in the loop variable, with the column names as keys. The value of the iterator style table attribute is ignored.

Please refer to the **`table blockloop`** command description for more information.

table dictloop

```
table dictloop tablehandle rowvariable ?rowobjectsvariable? ?maxrows? ?offset?
   body
t.dictloop(function=,?maxrows=?,?offset=?,?variable=?,?objectvariable=?)
```

This command is a variant of the **`table loop`** command. It stores the row data as a dictionary in the loop variable, with the column names as keys. The value of the iterator style table attribute is ignored.

Please refer to the **`table loop`** command description for more information.

table dup

```
table dup tablehandle ?rowrangelist? ?columnrangelist?
t.dup(?rows=?,?columns=?)
```

Duplicate a table. The return value is the handle or reference of the duplicate. Only the table data content is duplicated, not any table slices which refer to the original, or any ensembles or reactions which hold a reference to the original table. However, such references to ensembles or reactions are copied to the new table, so that the objects refer to both tables simultaneously after the copying. System tables can be duplicated just as any other table.

By default, the full table content is duplicated. The optional parameters can be used to restrict duplication to a row and/or column subset.

This command does not follow the standard *dup* command syntax of other major objects. It is not possible to move the duplicate table directly into a dataset.

Example:

```
set thnew [table dup $th]
```

table dupcolumns

```
table dupcolumns tablehandle columnrange ?position?
t.dupcolumns(columns=,?position=?)
```

Duplicate one or more columns, including their cell data, within a table. If the destination, a simple column address, is not specified, the duplicated columns are inserted on the right of the table. The destination cannot be in the range of the source columns.

The command may also be written as **`table dupcol`** or **`table dupcols`**.

The return value is the new number of columns.

table editcolumn

```
table editcolumn tablehandle column regexp ?substitution? ?flags?
t.editcolumn(column=,regexp=,?substitution?,?flags=?)
```

Edit the contents of a table column by performing regular expression matches and substitutions on cells which are not **`NULL`**. This also changes the data type of the column to *string*, which may have indirect consequences (see **`table setcol .. datatype`**).

If the substitution parameter is not supplied, the substitution value is an empty string, i.e. the matched part is removed from the cell data string.

The flags parameter may be a combination of the words

- *all* Substitute all matches of the regular expression. By default, it applies to the first match. Additional matches are only sought to the right of the last substitution, so this is not processed recursively.
- *nocase* Ignore case in regular expression matches
- *expanded* Support expanded regular expression syntax (see regexp man page)

`table editcol` is an alias of this command.

The command returns the original table handle or reference.

table ens

```
table ens tablehandle
t.ens()
```

Return a list of the handles or references of all ensembles which are referenced by the table. Every ensemble is reported only once, even if it is referenced by multiple rows. Rows with ensemble references are usually added to tables my means of the **`table addens`** command. In case a row has no ensemble reference, it is ignored, and no output is produced.

table exists

```
table exists tablehandle ?filterlist?
t.exists(?filters=?)
Table.Exists(tref=,?filters=?)
```

Check whether a table handle or reference is valid. The command returns boolean 0 or 1. Optionally, the table may be filtered by a standard filter list, and if it does not pass the filter, it is reported as not valid.

table expr

```
table expr tablehandle expression
t.expr(expression)
```

Compute a standard **SQL**-style property expression for the table. This is explained in detail in the chapter on property expressions.

table fill

```
table fill tablehandle rowrangelist ?columnrangelist? ?unsetonly? ?enshandle?
   ?reactionhandle?
t.fill(rows=,?columns=?,?unsetonly=?,?ens=?,?reaction=?)
```

Set the values of table data cells in the specified row range with data from an ensemble or reaction, for columns that are associated with a property or have another mechanism to compute or retrieve data from a chemistry object. If no column range is specified, the command visits all columns.

If the *unsetonly* flag is set, only **`NULL`** cells are modified and cells where valid data exists are skipped.

If an ensemble or reaction handle or reference is specified as argument, this object takes precedence over a potentially present cell object, or present row object. Otherwise, if there is a suitable cell object, it becomes the data source, and if there is none, the row object assumes this role. If there is also no row object, the command cannot perform any work and all selected cells remain unchanged.

The command can only modify cells in property columns, not pure data or formula columns. The property definition of the column is used to retrieve or compute the new cell data from the data source object. For reaction property columns, an explicit reaction object in the object priority sequence has precedence, and likewise for ensemble or ensemble minor object property columns, an explicit ensemble is preferred. If no directly matching source object is found, a reaction linked to the priority ensemble, or the first reaction ensemble from the priority reaction are used.

The command returns the original table handle or reference.

Note that for tables, there is no (it is rarely used, anyway) standard major object data manipulation command of the same name.

table filter

```
table filter tablehandle filterlist
t.filter(filters)
```

Check whether the filter passes a filter list. The return value is 1 for success and 0 for failure.

table find

```
table find tablehandle column|all operator value ?mode? ?retrievalcolumn?
t.find(column=,operator=,value=,?mode=?,?retrievalcolumn=?)
```

The command is very similar to the **`table select`** command. The difference is that this command stops the scan after the first matching row was found. Please refer to the section on that command for an explanation of the command arguments.

The command uses index information if the search operation allows it and a column index has been prepared. For large tables, this can make a big difference in search speed.

Example:

```
set r [table find $thandle E_CID = 5]
```

The result is the row index of the matching row, or minus one if no such row was found.

table flatten

```
table flatten tablehandle ?columnrange?
t.flatten(columns=)
```

This operation simplifies the data types represented in the table. **CACTVS** table columns can hold any data type the toolkit knows to manage via its data handler modules, including for example vector types, which are generally beyond the scope of traditional table formats.

This command attempts to simplify column types to elementary types, such as strings and numerics. In order to do that, columns with complex data types are split up. For example, a float vector column is replaced by a group of simple float columns, which are inserted immediately to the right of the original column. The number of these columns is determined by the longest vector found in the data cells under the original column, but it is at least one. The names of the new columns are either set to the names of the property fields (for example, for column data of type *compound*), or use the original column name with a bracketed suffix, for example `E_XYEXTENT(0)`. The first suffix is either zero or one, depending on the setting of the *offset* table attribute (see **`table get`**). The original column with the complex data is deleted. For columns which are already of a simple type, or of a type which is not handled by the current implementation, the command does nothing.

Currently, this function is only implemented for standard vectors (integer and float type vectors, bitvectors, plain and **UNICODE** string vectors) and the special data types *compound, choice, intpair, floatpair and intquad*. Other complex data types are not processed.

If no column range is specified, all table columns are processed. Table columns which are of an elementary data type are skipped.

The return value is the number of columns after the flattening operation.

table get

```
table get tablehandle propertylist ?filterset? ?parameterdict?
table get tablehandle attribute
t.get(property=,?filters=?,?parameters=?)
t.get(attribute)
t[property/attribute]
```

`t.property/attribute`

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For the use of the optional property parameter list and filter arguments, refer to the documentation of the **`ens get`** command.

In addition to retrieving property data, this command is also used to retrieve a large set of attribute values from the table object. Many of these attributes can also be set. Table objects have the following public attributes:

- *address_city*
  The city part of the author contact address.
- *address_country*
  The country part of the author contact address, following the ISO3166 standard.
- *address_state*
  The state part of the author contact address. Empty if not applicable.
- *address_street*
  The street address part of the author contact address. Includes floor, house number, etc.
- *address_zip*
  The **ZIP** code or other applicable postal code of the author contact address.
- *affiliation*
  The institution the table author works for.
- *affiliationduns*
  The **DUNS** registration ID of the affiliated institution. This is primarily useful for US government projects.
- *affiliationurl*
  The **URL** of the affiliated institution.
- *author*
  The author of the table, as free-form string data.
- *autorowformat*
  A string which is used as template for automatically generated table row names if no explicit names are specified. If it is not set, the fallback naming template is **`Row%ld`**, where the numeric placeholder is filled by the current value of the *autorowid* attribute. If a custom template is used, it should also contain a **`%ld`** placeholder, and no other C-style formatting instructions except for this one long integer.
- *autorowid*
  An integer which starts at zero when the table is created and is automatically incremented for every table row added via a **`table addrow`** command or internally called equivalents. This number is used in conjunction with the *autorowformat* attribute to generate unique table row names in case no explicit name is set. The value can be modified if necessary.
- *authorurl*
  A **URL** with information on the author, or an empty string if unset.
- *bgcolor*
  The global table background color. If unset, an the result is an empty string.

- *carbondisplaymode*
  The default mode for carbon atom rendering of embedded structure or reaction images. It can be either *none* (no symbols), *special* (special C atoms have symbols) or *all* (all carbon atoms are rendered with explicit symbols).
- *category*
  A category string to be used if the table is stored in a repository.
- *class*
  A class string to be used in **HTML** output as attribute of the **`<table>`** tag.
- *classuuid*
  The base class **UUID** of this table object
- *colblocksize*
  Set the column block size. If this value is larger than one, the default, the layout of some table output formats is adjusted to use repeated blocks of data instead of a simple matrix. For non-rotated table output, a new row is only started after the set number of entries, each with their normal item column count, have been written, instead of starting a new row after each physical table row. For rotated layout, a new printed set of rows, each set comprising of a number of rows equivalent to the physical table columns, is already forced after the specified number of rows have been printed left to right, instead of printing all selected rows left to right. This formatting option is currently only supported for Excel *xls*, HTML (table and page) and **CDXML** layout.
- *coldatatypes*
  The data type of the columns. If the list is shorter than the column count when used with a *set* command, only the first column data types are changed.
- *coldescriptions*
  A list of the column description texts.
- *collengths*
  The width of all columns as a list. Columns with undefined widths report minus one. This is a read-only attribute. Note that this attribute defines an object length (for example, a character count), not a formatting width, which is defined by the *widths* attribute.
- *colnames*
  The names of all columns as a list. If the list is shorter than the column count when used with a *set* command, only the first names are changed.
- *colproperties*
  A list of the properties of all table columns. If a column is not a property column, an empty string element is inserted. This is a read-only attribute. This attribute can also be addressed via its alias names *colprops, props* or *properties*.
- *coltypes*
  The column types of all columns as a list. This is a read-only attribute.
- *comment*
  A free-form comment string.
- *coords*
  If the toolkit was compiled with factory support, these are the coordinates of the object icon on its workbench, encoded as integer pair. This attribute can be changed.
- *dataset*
  A deprecated alternative name for the *internaldataset* attribute.

- *datatypes*
  A list of all the data types of the table columns. This is a read-only attribute. This attribute can also be addressed via its alias *coldatatypes*.
- *dataframe*
  This attribute is only available in the **PYTHON** interface and only if the **NUMPY** module is loaded or can be auto-imported. The complete table is exported as a **NUMPY** dataframe object. For column typing and special issues concerning these, please refer to the *nparray* and *nptype* column attributes.
- *date*
  The date the table configuration was defined.
- *deletable*
  Flag indicating whether the object can be deleted with a standard `table delete` command. This attribute is read-only. Objects which are, for example, property data values or a part of a `molfile loop` command cannot be deleted by standard means.
- *displaywidths*
  A list of the widths of all table columns. This is a read-only attribute. Table columns which have no explicit width contribute an empty string, not a negative or zero value. This is for compatibility with **Tk** widget configuration. Note that this is the formatting width, for example defined as pixel count or points, not an object length, such as a character count. The latter attribute can be queried by the *collengths* attribute. *widths* is an alias name for this attribute.
- *doi*
  A digital object identifier for the table object content, if defined.
- *downloadfilename*
  The save name of the file when it is transmitted via **HTTP**. If is is also specified as part of the transient output options in `table write`, that specification has precedence.
- *editable*
  Boolean flag reporting whether the table is editable.
- *email*
  A contact email of the author.
- *embedfileformat*
  The output file format of embedded objects in the table. This applies for example to tables which are written as `Excel` or **EXCEL XML**, where associated structures may be written as **CDX** or **SKC** `OLE` objects. For table output in formats which do not support this kind of embedding, the attribute is ignored. If the attribute is set to an empty string, or *none*, embedding is disabled where applicable, i.e. embedded **EXCEL XML** structure images are plain **WMF/EMF** drawings, not `OLE` objects. Otherwise, the attribute must be resolvable to a I/O module name for *molfile* objects (see `filex` command).
- *emptyrows*
  Get a list of row numbers which are empty, i.e. only contain `NULL` data values. This is a read-only attribute.
- *ens*
  A list of all ensembles associated with the table rows. This is a read-only attribute. If a structure data column is associated with the table, an attempt is made to automatically instantiate the ensembles.

- *ensrowcount*
  The number of rows which are associated with any ensemble. This is a read-only attribute. If a structure data column is associated with the table, an attempt is made to automatically instantiate the ensembles.
- *ensrows*
  A list of the row indices of those rows which are associated with an ensemble. This is a read-only attribute. If a structure data column is associated with the table, an attempt is made to automatically instantiate the ensembles.
- *eod*
  The value of the end-of-data marker. This attribute is typically used in multi-threaded applications to indicate that feeder threads have exhausted their data supplies and that no further table rows are expected to arrive. This attribute is internally used by the **`table pop`** and **`table wait`** commands to determine whether they should continue to wait or exit with an empty result. The initial value of this attribute is zero.
- *eodcheck*
  Perform a check whether at least one row is in the table, or is expected to arrive later. If rows are currently in the table, or the *eod* attribute value is less than the *targeteod* attribute value, the command returns zero, otherwise one. This attribute is read-only.
- *eolchars*
  The end-of-line chars for text-based table output. The default is platform-dependent (**`NL`** on Linux/Unix, **`CR`** on Mac, **`CR/NL`** on Windows).
  The magic strings *windows*, *mac* (both checked for the first three characters only) as well as *unix* and *linux* are automatically translated to the standard platform line terminators and not copied verbatim. Alternative names for these standard system encodings are *crlf*, *cr* and *lf*. The special value *default* resets the attribute to the platform-dependent default.
- *failures*
  A list of properties for which computation failed on this table object. This is a read-only attribute. Depending on configuration settings, this information may be used to block pointless attempts at re-computation of incomputable data.
- *fgcolor*
  The global table foreground color. If unset, an the result is an empty string.
- *fileformat*
  The format of the file the table was read from. *none* is returned if the table was not read from a file.
- *filename*
  The full path name of the file the table was read from. If the table was not read from a file, an empty string is returned. This attribute can also be set, and is then used when the table is written and no explicit filename given.
- *font*
  The global output font name, or an empty string if it is not set.
- *fontsize*
  The global font size in points. If not set, zero is reported.
- *footer*
  A free-form table footer text.

- *footercolor*
  The table footer color, or an empty string if not set.
- *footerfont*
  The table footer font name, or an empty string if not set.
- *footerfontsize*
  The table footer font size, or zero if not set.
- *footerformat*
  The table footer format flag set. The set is the same as for the format attribute.
- *format*
  The current set of global format flags. This is a keyword list which can contain the flags

  none -no flags, equivalent to empty string

  left - left-aligned text), *center* (centered text

  right - right-aligned text), *bold* (bold text

  highlight - cell highlight), *texthighlight* (text highlight

  histogram - plot histogram instead of data if supported by output format

  border - use extra cell border

  padding - use extra interior cell padding

  expand - format as auto-expand item

  top - text aligned vertically to top

  middle - text centered vertically in middle of cell

  bottom - text aligned vertically to bottom

  multiline - format cell data in multiple lines if possible

  gcolor - has explicit foreground color

  bgcolor - has explicit background color

  italic - use cursive text), underline (use underlined text

  embedded - embed content of data which are external references, such as linked files into the output, instead of passing the reference

  mdlnote - augment call data with MDL Molfile note text of associated ensemble or reaction

  smilesnote - augment cell data with SMILES note text of associated ensemble or reaction

  vertical - use vertical text if supported

  precision - use property precision data for formatting

  html - data is expected to be already HTML-encoded, no additional formatting for HTML table output

  merge - merge column with column on the left to multi-row column if supported

The flag set is the same in all contexts. However, not all make sense everywhere (i.e. the *merge* flag is only useful in column formats, not globally or on individual cells), and other require additional data (the *fgcolor*/*bgcolor* flags).

- *framecolor*
  The color of frames around objects in certain output contexts, for example around embedded images or OLE objects. This attribute is evaluated only if the rendering is executed implicitly, for example during an output operation. If, for example, an explicit `E_GIF` or `E_EMF_IMAGE` image is set as cell data, the parameters that were in effect when these images were generated are not checked. The default frame color is black. It is sometimes useful to set it to the background color or the rendering, i.e. white, or to suppress frame output altogether (see **`table write`**).
- *gflags*
  If the toolkit was compiled with factory support, this is the currently set object icon rendering flag collection.
- *header*
  A free-form header text.
- *headercolor*
  The table header color, or an empty string if not set.
- *headerdata*
  A list of the values of the header data for all columns. This is a read-only attribute.
- *headerfont*
  The table header font name, or an empty string if not set.
- *headerfontsize*
  The table header font size, or zero if not set.
- *headerformat*
  The table header format flag set. The flag set is the same as for the *format* attribute.
- *heights*
  A list of all row heights. This is a read-only attribute. Rows for which a height has not been set contribute an empty string.
- *hidden*
  Flag indicating whether the object is hidden. This is not the same as the *invisible* state. This attribute is intended to be used for rendering selections. This attribute can be changed.
- *highlightcolor*
  The default color for highlighting. It is set by default to red.
- *highlightfont*
  The default font for highlighted text output, or an empty string if not set.
- *highlightfontsize*
  The font size of the highlight font, or zero if not set.
- *highlightformat*
  The format flags used by default for highlighting. The set of flags is the same as for the format attribute. The default flag set is bold and underline (plus *fgcolor*, because the foreground color is set, see *highlightcolor* attribute).

- *httpheader*
  The type of **MIME** header to add to table output. It can be 0 (no header, the default), 1 (basic **MIME** header) or 2 (full header with status code).
- *hydrogendisplaymode*
  The default mode for hydrogen atom rendering of embedded structure or reaction images. It can be either *none* (no symbols), *special* (special H atoms have symbols) or *all* (all hydrogen atoms are rendered with explicit symbols).
- *imagedirectory*
  The name of a directory to store images and other external files in which, depending on the output format, cannot be stored directly in the table file.
- *imagemenuoptions*
  A keyword list to select the inclusion of specific image menu options in table file formats which support such a feature (currently, only **HTML**).
- *imageurl*
  The base **URL** component used for linking images in the image directory (attribute *imagedirectory*). In **HTML** output, this component and the name of image files were once combined to add working image links to the output file. Since in current toolkit versions, **HTML** output employs data **URI** encoding of the images as replacement for external linking, this attribute is deprecated.
- *infourl*
  A **URL** with information on the object content, or an empty string if unset.
- *internaldataset*
  The handle of the internal table dataset object which is part of every table. It is not the handle of a dataset the table may be a member of, as retrieved by the `table dataset` command. The attribute name may be shortened to simply *dataset*. It is a read-only value - it is not possible to change an internal table dataset since it is an integral part of the data structure.
- *invisible*
  Flag indicating whether the table is invisible. This is not the same as the *hidden* state. An invisible object is no longer accessible via its handle. This is usually the case for objects which are scheduled for deletion, but still have lingering referring pointers. This attribute is read-only.
- *iteratorstyle*
  The default style of row data stored in the loop variable of the `table loop` command. It can either be *list* (or *tuple*, for **PYTHON** compatibility) or *dictionary*. In *list* mode, the row data is presented as a simple list and elements are accessed via the list index. In dictionary mode, the data is presented as a *dictionary*, with the column names as keys. The default iterator style is *list*. More information can be found in the command description of `table loop.`
- *linktarget*
  If the table output is **HTML** or a similar format with hyperlinks, this parameter controls the link target specification, if it is not an empty string. For **HTML**, a value like *_blank* to open the link in a new window or the name of an existing window in an application can be set.
- *javaobject*
  If the toolkit was compiled with **JNI** support, this attribute reports the memory address of the **JNI** wrapper class instance, if it exists.

- *keywords*
  A list of keywords associated with the table object.
- *license*
  The license class associated with this table object. Setting the license to a standard type updates the associated **URL** with a standard location.
- *licenseurl*
  A **URL** with details about the table object license.
- *literature*
  A free-form literature reference for the data content of the table.
- *markcolor*
  The color of markings and annotations in certain output contexts, for example in embedded images or **OLE** objects. This attribute is evaluated only if the rendering is executed implicitly, for example during an output operation. If, for example, an explicit `E_GIF` or `E_EMF_IMAGE` image is set as cell data, the parameters that were in effect when these images were generated are not checked. The default mark color is pure blue.
- *maxrows*
  A maximum number of rows allowed in the table. Attempts to add more rows fail, or, if the table has an active background export thread, block. Setting it to a negative value, the default, disables this limit.
- *modcount*
  The object data and content modification count.
- *mutexcount*
  The number of recursive mutex locks held for this object. Only supported on Linux.
- *name*
  A free-form table name as string
- *ncols*
  The number of columns in the table. This attribute may be used in a **`table set`** command to reduce the number of columns, but not to enlarge the table.
- *note*
  Arbitrary data added as note data. This is usually only used on a lower level to added notes to individual cells.
- *notesize*
  The size of the table-level note data. This is a read-only attribute.
- *nrows*
  The number of rows in the table. This attribute may be used in a **`table set`** command to reduce the number of rows. Adding table rows is also possible by manipulating this attribute. In that case, all new data cells are set to **`NULL`**.
- *nulldatarows*
  A list of the row indices of those rows where all columns are **`NULL`** values.
- *nullensrows*
  A list of the row indices of those rows which are not associated with an ensemble. This is a read-only attribute. If a structure data column is associated with the table, an attempt is made to automatically instantiate the ensembles.

- *nullreactionrows*
  A list of the row indices of those rows which are not associated with a reaction. This is a read-only attribute. If a reaction data column is associated with the table, an attempt is made to automatically instantiate the reactions.
- *offset*
  This attribute can be either zero or one. It influences the naming of flattened columns (see **`table flatten`** command)
- *onclick*
  For **HTML** output, the name of a JavaScript function to be called when a table cell is clicked. The function is called with three arguments: The automatically generated **`DOM ID`** of the table cell, and the row and column indices, starting with zero.
- *onmouseover*
  For **HTML** output, the name of a JavaScript function to be called when a the mouse is moved over a table cell. The function is called with three arguments: The automatically generated DOM ID of the table cell, and the row and column indices, starting with zero.
- *onmouseout*
  For **HTML** output, the name of a JavaScript function to be called when a the mouse is moved away from a table cell. The function is called with three arguments: The automatically generated **DOM** ID of the table cell, and the row and column indices, starting with zero.
- *orcid*
  The **ORCID** code of the author of the table (see www.orcid.org).
- *orientation*
  This value can be *none* (the default), *landscape* or *portrait*. It describes the orientation of a drawing area specified via the paper attribute. Few I/O modules use this information. The most important formats which implement this is are **CDX** and **CDXML**.
- *paper*
  An attribute describing the size of the drawing areas for formats such as **CDX** or **CDXML**, which can encode this type of information. Possible values are *none* (the default), *a3*, *a4*, *a5, a6, a7, b3, b4, b5, b6, letter, legal* and *executive*. The associated orientation of the drawing orientation can be set via the *orientation* attribute
- *parameters*
  A keyword/value dictionary of format-specific output options which are not represented by a general table object attribute. I/O modules may possess individual additional control parameters (see **`tablex get/set`** commands). The default values for these are automatically added to the *parameters* table attribute when a table file is written, and may be examined afterwards, but key/value pairs which have been explicitly configured via this attribute are not overwritten and take precedence.
- *path*
  The repository path for displaying hierarchical repository trees. This attribute is independent of any file system paths.
- *phone*
  A contact phone number of the author.

- *progress*
  A user-defined progress value intended to track the state of lengthy operations on the table. It is an integer between zero and one hundred and is initially set to zero. When the argument is set, it accepts a floating point value, but the stored value is automatically rounded to the next integer and forced into the 0..100 range.
- *pyobject*
  If the toolkit was compiled with Python support, this attribute reports the memory address of the Python wrapper class instance, if it exists. This attribute is read-only.
- *pyrefcount*
  If the toolkit was compiled with Python support, this attribute reports the reference count of the Python wrapper class instance, if it exists. This attribute is read-only.
- *reactioncolumn*
  The index of the first column which has the *reactionsource* attribute set. If no such column can be found, the result is minus one. This attribute is read-only.
- *reactioncolumnfileformat*
  The file format associated with columns which contain decodable reaction data. By default, this attribute is unset, which means that an attempt is made to automatically detect the format, as with a **`reaction create`** command.
- *reactionimageproperty*
  The name of a property to use for rendering reaction images in formats which allow the inclusion of images in multiple formats, such as **HTML**. In that case, supported properties include `X_GIF`, `X_SVG_IMAGE` and `X_PDF_IMAGE`. In table file formats which cannot include images, or support only a single image format, this attribute is ignored.
- *reactions*
  A list of all the reactions associated with the table rows. This is a read-only attribute.
- *reactionrowcount*
  The number of rows which are associated with a reaction. This is a read-only attribute. If a reaction data column is associated with the table, an attempt is made to automatically instantiate the reactions.
- *reactionrows*
  A list of the row indices of those rows which are associated with a reaction. This is a read-only attribute. If a reaction data column is associated with the table, an attempt is made to automatically instantiate the reactions.
- *record*
  The current table row iterator record. The first row is record one.
- *refcount*
  If the **Tcl** interpreter is using native **Cactvs** objects instead of string-based major object handles and integer-based minor object labels to identify toolkit objects, this returns the number of **Tcl** object references active for this table. This attribute is read-only.
- *references*
  Cross references of the table. This is a nested list of class **UUID**s and reference type tags.
- *regid*
  For registered data tables, the registration ID. Zero if this is a private table.

- *rownames*
  A list of the names of all rows. This is a read-only attribute.
- *rownametype*
  This can be either *string*, or *numeric*, and has an effect on how row names are compared during sorting and other operations. The names are nevertheless always stored as strings.
- *rowobjectassociations*
  This is a boolean attribute. On querying, it indicates whether this table has any rows which are associated with existing ensemble or reaction objects. When setting, it has an effect only if the boolean argument value is **`false`**. This operation then removes any existing associations between structures or reactions and rows on this table (see **`table addens/addreaction`**). Deleting structures or reactions while they are associated with a table can be slow if multiple or large tables exist in the application, since they all need to be searched for associations with the object to be deleted. This is because currently structures or reactions only maintain a table reference count, but not any information with which table(s) and row(s) they are associated with.
- *scoped*
  A boolean object visibility control flag. If set, and global control flag **`::cactvs(object_scope)`** is also set, the object is visible only in the**TCL** interpreter which set the scope flag and thus claimed it. Object list commands executed in other interpreters omit this object, and attempts to decode its handle in other interpreters will fail. The most common use of this feature is the hiding of persistent chemistry objects in scripted property computation functions.
- *selected*
  Flag indicating whether the object is selected. This attribute can be changed. This flag is unrelated to row or column selections - it is a standard major object flag.
- *selectedcolumns*
  The currently selected set of columns, as a numerical list of column indices.
- *selectedrows*
  The currently selected set of rows, as a numerical list of row indices.
- *separator*
  The current separator character. This is updated if table data is read from a text file with automatic detection of the separator character or with an explicitly named separator (see **`table read`** command). It also has an effect on formatting of ASCII table output. The default separator is a tab character. If you set it to a string which has more than one character, only the first character is actually recorded.
- *sizehint*
  An integer indicating the expected maximum row count of the table. This is used for internal optimizations. A value of minus one indicates that no size hint is in effect.
- *soapmethod*
  The name of a method which is used for **SOAP** data exchange involving this table.
- *soapschema*
  The name of a schema which is used for **SOAP** data exchange involving this table.
- *sqlddl*
  Compose an **SQL** statement which can be used to define a compatible database table. The precise syntax used depends on the configured **SQL** dialect in **`::cactvs(sql_dialect)`**.

- *structurecolumn*
  The index of the first column which has the *structuresource* attribute set. If no such column can be found, the result is minus one. This attribute is read-only. The attribute name *enscolumn* is an alias.
- *structurecolumnfileformat*
  The file format associated with columns which contain decodable structure data. By default, this attribute is unset, which means that an attempt is made to automatically detect the format, as with a **`ens create`** command. The attribute name *enscolumnfileformat* is an alias.
- *structureimageproperty*
  The name of a property to use for rendering structure (ensemble) images in formats which allow the inclusion of images in multiple formats, such as **HTML**. In that case, supported properties include `E_GIF`, `E_SVG_IMAGE` and `E_PDF_IMAGE`. In table file formats which cannot include images, or support only a single image format, this attribute is ignored.
- *style*
  The name of a predefined output style. This is currently only used by the **CDX** and **CDXML** table output formats, which support both the **ChemDraw** default style and the ACS journal style (*acs*).
- *stylesheet*
  This is only used in *htmltable* output. It is the outputfile-relative path name of a **CSS** file which is linked to the primary output file via a **`<link>`** tag.
- *symboldisplaymode*
  The default mode for atom rendering of embedded structure or reaction images. It can be either *none* (no symbols), *symbol* (element symbols), *label* (atom labels), *index* (atom list index), *box* (colored square marker), *compact* (symbols with contracted atoms) or *residue* (`A_RESIDUE` label)
- *targeteod*
  The target value of the *eod* attribute. Once it matches or exceeds this value, the table is not expected to receive any more rows. The initial value of this attribute is one. This attribute is for example checked by processing threads.
- *title*
  A free-form table title text.
- *titlecolor*
  The table title color, or an empty string if not set.
- *titlefont*
  The table title font name, or an empty string if not set.
- *titlefontsize*
  The table title font size, or zero if not set.
- *titleformat*
  The table title format flag set. The set is the same as for the *format* attribute.
- *tooltip*
  If the toolkit was compiled with factory support, this is the tooltip of the object icon on a workbench. This attribute can be changed.

- *undefined*
  This attribute controls the I/O of undefined cell data. It is a string consisting of one or more words, separated by commas. The first word is used for string-based output of **NULL** cell values. All words are recognized when decoding cell data from strings. Any input matching any of the words are stored as a **NULL** value. The default value is „*NULL,N/A,UNDEF,UNDEFINED,UNSET,NOTSET*“. With this string, **NULL** cells are printed as “NULL”, and a total of six magic words for the input of **NULL** data are recognized.
- *uuid*
  An automatically generated **UUID** globally identifying the object. This attribute is read-only, different for every object, and not dependent on its contents.
- *variable*
  The name of a global **Tcl** array variable in the base interpreter which is linked to the table. Table updates are reflected in the variable contents, and changes of the variable elements will change the table object contents. The 2-dimensional **Tcl** array variable uses numerical row and column indices. If there is no linked variable, which is the default, the result is an empty string.
- *version*
  The version number of the table. This is a string in a 1.2.3 (or shortened) style,
- *versionuuid*
  The **UUID** associated with this table object version.
- *x*
  If the toolkit was compiled with factory support, this is the x coordinate of the object icon on its workbench. This attribute can be changed.
- *y*
  If the toolkit was compiled with factory support, this is the y coordinate of the object icon on its workbench.This attribute can be changed.

Table attributes which are not marked read-only can be set by the **`table set`** command.

Variants of the **`table get`** command are **`table new`**, **`table dget`**, **`table jget`**, **`table jnew`**, **`table jshow`**, **`table nget`**, **`table show`**, **`table sqldget`**, **`table sqlget`**, **`table sqlnew`**, and **`table sqlshow`**. These commands only work on property data and cannot be used to access attributes.

table getcell

```
table getcell tablehandle row column ?attribute?
t.getcell(row=,column=,?attribute=?)
```

Get the cell data value, or a cell attribute. The following cell-level attributes are supported:

- *bgcolor*
  The background cell color, if it is set on the cell level.
- *comment*
  A free-form text comment
- *exists*
  Check whether the addressed cell exists in the table. The return value is a boolean result. No error is raised if the cell address cannot be resolved.

- *fgcolor*
  The foreground cell color, if it is set on the cell level.
- *format*
  Cell-level format flags. This is the same set as for the global table attribute of the same name, which is explained in the section describing the `table get` command.
- *isnull*
  A boolean check whether the data value of the cell is **NULL**.
- *mergedbgcolor*
  The effective cell background color, merged from cell>row>column>table format specifications.
- *mergedfgcolor*
  The effective cell foreground color, merged from cell>row>column>table format specifications.
- *mergedformat*
  The effective format flags for the cell, merged from cell>row>column>table format specifications.
- *image*
  A **Tk** image handle when the table object is linked to a **Tk** table widget.
- *note*
  A free-form cell note. In output formats such as **MS Excel** this is displayed as a tool tip. Note that this data can be binary.
- *notesize*
  The size of the note, in bytes.
- *object*
  The table cell object (the handle of an ensemble or reaction), or an empty string if no such object exists.
- *tag*
  The cell tag in a linked **Tk** table widget.
- *value*
  The cell data value.
- *window*
  The name of a linked **Tk** window displaying cell contents.

The attributes *image, tag* and *window* are only useful for developers who want to display a **Cactvs** table in a **GUI** tool developed with the **Tk** toolkit and its table widget.

If the last optional command argument is omitted, it is assumed to be *value*.

`table cellget` is an alias for this command.

table getcolumn

```
table getcolumn tablehandle column ?attribute?
```

`t.getcolumn(column=,?attribute=?)`

Get data from a specific column. All attributes which can be set (see **`table setcolumn`** command), can also be read. In addition, the following attributes can only be read, but not set:

- *avg*
  The computed average value of the column data cells. **NULL** cells are ignored. This operation only succeeds on numerical table columns. *average* is an alias.
- *count*
  The number of non-null data values under this column.
- *data*
  A list of all the values of the column data cells, in row order. The size of the returned list is the same as the row count. *values* is an alias.
- *data(fieldname)*
  This form is a variant of above command. It is used to access a specific data field of the column data, or perform datatype-specific conversion operations. The allowed field names are determined by the specification of the column property, if it exists, and/or the column datatype. If the column data is already a property field, the field name signifies further indexing from that property data subset onwards. At this time, not all indexing operations possible in other index-driven commands are available, especially index accesses where individual data items determine success or failure (such as key-indexed dictionary values) are not possible.
- *distinct*
  Get the number of distinct cell values in this column. Implicitly, this sets up a column index.
- *exists*
  A boolean pseudo-attribute indicating whether the column with the specified identifier exists in the current table or not.
- *hasnull*
  A boolean flag indicating whether any of the cells in the column is a **`NULL`** cell.
- *index*
  The numerical column index of the specified column identifier. If the identifier cannot be resolved, minus one is reported.
- *isnull*
  A boolean value indicating whether all rows under the column exclusively contain **`NULL`** values.
- *max*
  The computed maximum value of the column data cells. **NULL** cells are ignored. This operation only succeeds on numerical table columns.
- *min*
  The computed minimum value of the column data cells. **NULL** cells are ignored. This operation only succeeds on numerical table columns.

- *maxsize*
  The the maximum size (as per the *size* pseudo-element in property retrieval commands) of all non-null rows of the column. This is especially useful for string-class columns (returning the maximum string length) and vector-class columns (maximum vector size). If there are no non-null cells, in the column, the return value is an empty string.

- *minsize*
  The the minimum size (as per the *size* pseudo-element in property retrieval commands) of all non-null rows of the column. This is especially useful for string-class columns (returning the minimum string length) and vector-class columns (minimum vector size). If there are no non-null cells in the column, the return value is an empty string.

- *nparray*
  This attribute is only available in the **Python** interface and only if the **numpy** module is loaded or can be auto-imported. The return value is a **numpy** array of the column data. Vector data is either represented as 2-dimensional arrays (if the vector length is constant over all column values), or *object_* if it varies. Not all **Cactvs** data types can be translated to **numpy**, and the command can thus fail for more complex column data types. For an export of a full table as a **numpy** dataframe, see the *dataframe* attribute on the table object.

- *nptype*
  This attribute is only available in the **Python** interface, and only if the **numpy** module is loaded or can be auto-imported. It returns the **numpy** item datatype for elements of this column as **numpy** object, or `none` if the column data cannot be represented as **numpy** data. For vectors the type is the element type if all vector lengths in the table column are identical, otherwise *object_*, due to limitations in the **numpy** data model.

- *nullrows*
  Get a list of all rows which have `NULL` values in this column.

- *notesize*
  The byte length of the *note* attribute. If no note is attached, the result is zero.

- *objects*
  A list of the handles of all table cell objects set in that column, in row order. Cells for which no object has been set report an empty string list element.

- *propertyfield*
  The property field index of the column. If the column is not a property column, or the column is specified to contain data of a complete property record, not just a field, the value minus one is returned.

- *sqlname*
  The name of the column, reformatted to be easily usable in the context of SQL statements. These column names are automatically used when the table is exported in one of the SQL database dialects.

- *stddev*
  The computed standard deviati8on of the column data cells. **Null** cells are ignored. This operation only succeeds on numerical table columns.

- *sum*
  The computed sum of the column data cells. **NULL** cells are ignored. This operation only succeeds on numerical table columns.
- *unique*
  Test whether the row values under this column are unique. The command returns a list of three values. The first is a boolean value telling whether all values are unique, followed by a pair of row indices of a duplicate value pair if it exists. If the column values are unique, the reported row indices are both minus one. The uniqueness test is performed by temporarily creating a column index and thus scales better than n^2, but is also means only columns where an index can be built can be tested.

The *exists* and *index* attributes allow the specification of non-existing column identifiers. In all other cases, an error is raised if the column cannot be identified.

If the last optional command argument is omitted, it is assumed to be *data*.

`table colget` and **`table getcol`** are aliases for this command.

Example:

```
table colget $th mycol
table colget $th mycol data(CHF)
```

The first form retrieves all column data values in standard format. The second form assumes that the column supports index *CHF*, which is for example the case if the column data type is *currency*. In that case, the returned values are automatically converted from whatever original currency it is in.

table getparam

```
table getparam tablehandle property ?key? ?default?
t.getparam(property=,?key=?,?default=?)
```

Retrieve a named computation parameter from valid property data. If the key is not present in the parameter list, an empty string is returned (**None** for **PYTHON**). If the default argument is supplied, that value is returned in case the key is not found.

If the key parameter is omitted, a complete set of the parameters used for computation of the property value is returned in dictionary format.

This command does not attempt to compute property data. If the specified property is not present, an error results.

table getrows

```
table getrows tablehandle rowrange ?attribute?
t.getrows(rows=,?attribute=?)
```

Get data from a specific row *range*. All attributes which can be set (see **`table setrow`** command), can also be read. In addition, the following attributes can only be read, but not set:

- *data*
  A nested list of all the values of the data cells of the row set. The size of the returned inner list is the same as the column count, and the size of the outer list the row count between the first and last specified rows. *values* is an alias.

- *dictdata*
  The same as *data*, except that the row information is returned as a dictionary, with the column names as keys. *dictvalues* is an alias.
- *exists*
  Report as a boolean value whether the specified row range could be resolved.
- *index*
  The list of numerical row indices of the specified row identifiers. If the row range cannot be resolved, a single minus one value is reported.
- *isnull*
  A boolean value indicating whether all columns in the specified row range consist exclusively of **`NULL`** values.
- *objclass*
  The class of objects, such as *ens*, *reaction* or *atom*, associated with each row in the retrieval range. The default attachment, even for rows which are not connected to a chemistry object, is *ens*.
- *objects*
  A nested list of the cell objects in that row set. Cells for which no cell object has been set report an empty string.
- *notesize*
  A list of the he byte lengths of the *note* attributes in the row set. If no note is attached, the result is zero for that row.

If the last optional command argument is omitted, it is assumed to be *data*.

`table rowget` and **`table getrow`** are aliases for this command.

A common error when using this command is to forget that it can return data on multiple rows, and therefore the return value is always nested list. If only a single row needs to be accessed in a statement, and list unwrapping is tedious, the **`table getrow1`** command can be used instead.

table getrow1

```
table getrow1 tablehandle row ?attribute?
t.getrow1(row=,?attribute=?)
```

This command is essentially the same as **`table getrows`**, but it only accepts a simple single-row identifier, not a row range.

The return value is a simple list for the column values, not a nested list as with **`table getrows`**, which can be convenient.

Important: For reasons of backward compatibility, **`table getrow`** is an alias for **`table getrows`**, not **`table getrow1`**!

table getparam

```
table getparam tablehandle property ?key? ?default?
t.getparam(property=,?key=?,?default=?
```

Retrieve a named computation parameter from valid property data. If the key is not present in the parameter list, an empty string is returned (`None` for **PYTHON**). If the default argument is supplied, that value is returned in case the key is not found.

If the key parameter is omitted, a complete set of the parameters used for computation of the property value is returned in dictionary format.

This command does not attempt to compute property data. If the specified property is not present, an error results.

table hierarchy

```
table hierarchy thandle ?filterlist? ?root?
t.hierarchy(?filters=?,?root=?)
```

Return the hierarchy handle or reference of the hierarchy the table is part of. If the table is not member of a hierarchy, or does not pass all of the optional filters, an empty string or `None` for **PYTHON** is returned. By default, the hierarchy object which directly contains the table is returned. If the *root* flag is set, the root hierarchy object is reported instead, which is the same only if the hierarchy has only a single level.

Example:

```
table hierarchy $thandle
```

table index

```
table index tablehandle
t.index()
```

Get the position of the table in the object list of its dataset. If the table is not member of a dataset, -1 is returned.

table innerjoin

```
table innerjoin tablehandle1 tablehandle2 ?column1? ?column2? ?cmpflags?
table join tablehandle1 tablehandle2 ?column1? ?column2? ?cmpflags?
t.innerjoin(table2=,?column1=?,?column2=?,?comparisonflags=?)
t.join(table2=,?column1=?,?column2=?,?comparisonflags=?)
```

Perform a relational inner join on two tables and return the handle or reference of a newly created join table. The two command variants are aliases.

The source tables remain unchanged. If no explicit table columns are specified, in both tables the first column is used. If only one column name or index is specified, it applies to both tables. With two column names or indices, the first is used to select the join column on the first input table, and the second on the other table.

This command explanation is also referenced by the table object subcommands *leftjoin*, *rightjoin*, and *outerjoin*. This is the reason why the explanation of some command features goes beyond describing exclusively the *innerjoin* subcommand.

The result table always contains, in this order, the columns of the first table, followed by those of the second table, with the exception of the omitted join column of the second table. The names of columns originating from the second table are adjusted if necessary to avoid duplication of column names in the result table. Data attached to rows on either input table outside of cell values, such as structure or reaction references, are not copied to the output table.

By default, the join condition is a simple equality check on the data of a single column from each table. The data value comparison method may be modified by the last optional argument, for example to use case-insensitive or punctuation-ignorant comparison. The recognized values in this flag set are dependent on the column data types and the same as in the **`prop compare`** command. At this time, the fundamental comparison operator is always an equality check - other operators such as *less than* etc. are not supported. The row values of the join columns are not required to be sorted in either input table. **`NULL`** data values in the comparison column never match another data value, even if it also is **`NULL`**, following the standard rules of relational algebra. Parallel joining on multiple columns is not yet supported.

The column data types are not required to be identical. If they disagree, an attempt is made to cast the data values of the first table to the datatype of the column of the second table. If that fails, the input row of the first table is skipped and never included in the result table, even in left, right or outer joins.

In an inner join, rows from the first able are omitted in the output if there is no matching column value in the second table. This is always the case if the column value is **`NULL`** - a **`NULL`** value is never equivalent to another **`NULL`**. If either table has more than one matching row for a column value, multiple combined output rows result from that pairing, with all possible combinations of row data blocks (but not individual join column values - these are by definition identical) from either table result.

The **`table leftjoin`**, **`table rightjoin`** and **`table outerjoin`** commands are extensions of the fundamental inner join operation. These generate selected additional rows in the output for rows from the tables which do not match a counterpart in the join column, following the normal relational algebra rules.

table jget

```
table jget tablehandle propertylist ?filterset? ?parameterdict?
t.jget(property=,?filters=?,?parameters=?)
```

This is a variant of **`table get`** which returns the result data as a **JSON** formatted string instead of **TCL** or **PYTHON** interpreter objects. The command is usable only for property data, not attribute retrieval.

table jnew

```
table jnew tablehandle propertylist ?filterset? ?parameterdict?
t.jnew(property=,?filters=?,?parameters=?)
```

This is a variant of **`table new`** which returns the result data as a **JSON** formatted string instead of **TCL** or **PYTHON** interpreter objects.

table jshow

```
table jshow tablehandle propertylist ?filterset? ?parameterdict?
t.jshow(property=,?filters=?,?parameters=?)
```

This is a variant of **`table show`** which returns the result data as a **JSON** formatted string instead of **TCL** or **PYTHON** interpreter objects.

table leftjoin

```
table leftjoin tablehandle1 tablehandle2 ?column1? ?column2? ?cmpflags?
t.leftjoin(table2=,?column1=?,?column2=?,?comparisonflags=?)
```

Perform a relational left join on two tables and return the handle or reference of a newly created join table. The command arguments and result value are explained in the **`table innerjoin`** command.

The difference between a standard inner join and a left join is that rows from the first input table which do not match any values of the join column in the second table are still output, with all column data originating from the second table set to **`NULL`** values.

table list

```
table list ?filterlist?
Table.List(?filters=?)
```

Without a filter list argument, the command returns a list of the handles of all tables currently existing in the application, including those of the system tables.

If a filter list is specified, only those tables which pass all filters are listed.

Examples:

```
table list
```

table listblockloop

```
table listblockloop tablehandle rowvariable ?maxrows? ?offset? body
t.listblockloop(column=,function=,?maxblocks=?,?offset=?,?variable=?)
```

This command is a variant of the **`table blockloop`** command. It stores the row data as a simple nested list in the loop variable, without column names. The value of the iterator style table attribute is ignored.

Please refer to the **`table blockloop`** command description for more information.

For the sake of compatibility with the **Python** interface syntax, the command may also be invoked as **`table tupleblockloop`**.

table listloop

```
table listloop tablehandle rowvariable ?rowobjectsvariable? ?maxrows? ?offset?
   body
t.listloop(function=,?maxrows=?,?offset=?,?variable=?,?objectvariable=?)
```

This command is a variant of the **`table loop`** command. It stores the row data as a simple list in the loop variable. The value of the iterator style table attribute is ignored.

Please refer to the **`table loop`** command description for more information.

For the sake of compatibility with the **Python** interface syntax, the command may also be invoked as **`table tupleloop`**.

table lock

```
table lock tablehandle propertylist/table/all ?compute?
t.lock(property=,?compute=?)
```

Lock property data of the table, meaning that it is no longer subject to the standard data consistency manager control. The data consistency manager deletes specific property data if anything is done to the table which would invalidate the information. Blocking the consistency manager can be useful when building tables from components in a script. Property data remains locked until is it explicitly unlocked.

The property data to lock can be selected by providing a list of the following identifiers:

- Property names
  Valid property instances on the table are locked. If the boolean *compute* flag is set, an attempt is made to compute the property if it is not yet present. Otherwise, a request to lock non-existent data is silently ignored. It is not possible to lock individual property fields.
- *all*
  All valid table properties are locked. The compute flag is ignored.
- *table*
  This is an object class identifier. All property data which is controlled by the table major object and attached to the specified object class is locked. Because a table currently does not have minor objects, this command does the same as the *all* variant.

The lock can be released by an **`table unlock`** command.

The return value is the table handle.

table loop

```
table loop tablehandle rowvariable ?rowobjectsvariable? ?maxrows? ?offset? body
t.loop(function=,?maxrows=?,?offset=?,?rowvariable=?,?objectsvariable=?)
for r in t:
```

This command executes a loop over the rows of a table. On each iteration, the variable *rowvariable* is set to a list of the cell values of the current row, and then the *body* argument is executed as script.

For **TCL** scripts, within the loop, the standard **TCL** **`break`** and **`continue`** commands work as expected. If the body script generates an error, the loop is exited.

By default, the loop runs from the first row to the end of the table. The optional arguments can be used to set a maximum iteration count, and to change the starting point of the loop. If the maximum iteration count is explicitly set to a negative value, the loop runs to the end of the table.

The content of the row variable is dependent on the configured table iterator style (see *iteratorstyle* table attribute). It can either be a list, with simple cell values, or a dictionary with column name and cell values. The default is the list style.

If a row objects variable is specified, it contains the handles of the cell objects in the current row, and empty strings (**`None`** for **PYTHON**) for those cells without a cell object. The object variable cannot be a string looking like an integer, otherwise the argument will be interpreted as *maxrows* argument. It is also possible to explicitly skip the argument by using an empty variable name. The reported cell objects are direct references to the cell content and remain locked to the table. They cannot and do not need to be discarded in the loop.

During the execution of the loop, the *record* table attribute is continuously updated to the current 1-based row number.

The **Python** version of the loop method does intentionally have a different argument sequence for convenience. The function argument may either be a multi-line string (similar to the **Tcl** construct), or a function reference. Functions are called with the table handle, 0-based row number, row data tuple or dictionary and the row objects tuple or dictionary as four arguments.

Normal **Python** functions have their own context frame, and are passed arguments, so that the specification of a loop variables is not generally useful in that call style, though is is allowed. For string function blocks the code is executed in the local call frame, and the variable with the current object reference is visible locally. Script code blocks must be written with an initial indentation level of zero. Within the **Python** functions, the normal *break* and *continue* loop control commands cannot be used to to scope limitations. Instead, the custom exceptions *BreakLoop* and *ContinueLoop* can be raised. These are automatically caught and processed in the loop body handler code.

In **Python**, there is also an object iterator so that simple loops over table rows, with the row data as iterator value, can be written with a **`for`** statement. The table object iterator is of the *self* style (i.e. there is one per table, these are not independent objects), so nesting them is not possible on the same table.

Python object loop constructs and their peculiarities are discussed in more detail in the general chapter on **Python** scripting.

Example:

```
table loop $th rowvar {
   echo "Col0: [lindex $rowvar 0] Col1: [lindex $rowvar 1]"
}
```

The **`table blockloop`** command is a more complex variant of this command. It operates on blocks of rows with a common cell value in a column instead of simply stepping from one row to the next.

The **`table listloop`** variant of this command always stores the row data as a list in the loop variable. The value of the iterator style table attribute is ignored.

The **`table dictloop`** variant of this command always stores the row data as a dictionary in the loop variable, with the column names as keys. The value of the iterator style table attribute is ignored.

There are also *rownamedictloop* and *rownamelistloop* variants, which additionally report the name of the current table row.

The return value of the command is the number of loop iterations processed. The last value of the loop variables remain accessible outside the loop, if loop variables were used.

table merge

```
table merge tablehandle1 tablehandle2 ?rowmode? ?columnmode? ?comparisoncolumn?
t.merge(table2=,?rowmode=?,?columnmode=?,?comparisoncolumn=?)
```

Merge two tables into one. The first table is updated. The second table remains valid and retains all its data. It is not possible to merge a table with itself.

The first table will usually be subject to row and column updates. What kind of updates are performed is determined by the row and column modes. The row mode can be one of:

- *append*
  All rows from the second table are appended to the first, without checking for duplicates.

- *exclusive*
  Any rows which are present in both tables are deleted from the first table. In addition, rows from the second table which do not match rows in the first table are appended. This is the opposite of the *intersect* mode.
- *intersect*
  Rows from the first table which have no equivalent in the second table are deleted from the first table. Matching rows remain unchanged in the first table, and no data is ever appended to the first table.
- *join*
  This mode is a combination of the *transfer* and *union* modes.
- *replace*
  The data of rows which have a duplicate in the first table overwrites all data cells in the matching row in the first table. This can be used, depending on the column mode, to add or update column data which is not present or **`NULL`** in the first table. The difference to the *transfer* mode is that even non-**`NULL`** cells in the first table are overwritten by data from the second table. Rows from the second table which do not match a row in the first table are ignored.
- *second*
  Rows in the second table which do not match any rows in the first table are appended to the first table. All original rows in the first table are deleted. This is similar to the *subtract* mode, with the order of the tables reversed.
- *subtract*
  Only rows from the first table which do not match rows from the second table are retained in the first table. Matching rows are deleted. No data is appended.
- *transfer*
  The data of rows which have a duplicate in the first table overwrites **`NULL`** data cells in the matching row in the first table. This can be used, depending on the column mode, to add or update column data which is not present in the first table. The difference to the replace mode is that non-**`NULL`** cells in the first table are not overwritten by data from the second table. Rows from the second table which do not match a row in the first table are ignored.
- *union*
  Rows from the second table for which there is no duplicate in the first table are appended to the first table. This is the default row mode. Rows which are duplicates are skipped.

The *comparisoncolumn* parameter determines which column is used to check for duplicate rows. By default, it is *#name*, i.e. the row name is used instead of real column data. For tables without explicit row names, this is equivalent to the row index. For tables which were generated by direct output from the **`molfile scan`** command, it has been automatically set to the matched file record number. Query result tables from the same file can therefore be merged easily. Instead of the row name, any cell data of a real column in the first table may be used for identity checks. However, this column must have an equivalent in the second table, or an error results.

The column mode determines which column definitions from the second table are added to the first table, and whether any columns on the first table are deleted if there is no corresponding column in the second table. The supported column modes are:

- *append*
  All columns of the second table are appended to the right of the current table. This can lead to duplicate column names and is not a frequently used mode. Old row data on the first table becomes **`NULL`** for these new columns.
- *left*
  The columns on the first table are not changed.
- *intersect*
  All columns on the first table for which no equivalent column exists in the second table are deleted in the first table. Columns in the second table without an equivalent in the first table are ignored and their cell data will not be copied from the second table. This is the default column merge mode.
- *right*
  The second table determines the column set. Any column on the first table for which there is no equivalent in the second table are deleted, and any column definitions in the second table for which there is no equivalent in the first table are added to the first table. Old row data on the first table becomes **`NULL`** for these new columns.
- *union*
  Any columns on the second table for which no equivalent column exists are recreated on the first table. Old row data on the first table becomes **`NULL`** for these new columns.

The order of the columns on the two merged tables is not important for property columns. Column matching uses property definitions and field indices to find equivalent columns regardless of their name. Pure data columns without property definitions require a matching name and data type in order to be perceived as equivalent. The column modes *left*, *intersect* and *union* are most often used. If the two tables have identical column sets, the merge result are the same in all these modes.

If property `T_QUERY` is valid on both tables, and it describes a query on the same data source, it is updated on the first table to represent a single query which would yield the results of the merged table. This is done by manipulating the query tree in that property by combining the old two trees under a suitable logical operator such as *and* or *or*. This is intended to facilitate convenient management of query results in tables, as they are, for example, produced by a **`molfile scan`** command with direct table output.

table metadata

```
table metadata tablehandle property ?field ?value??
t.metadata(property=,?field=?,?value=?)
```

Obtain property metadata information, or set it. The handling of property metadata is explained in more detail in its own introductory section. The related commands **`table setparam`** and **`table getparam`** can be used for convenient manipulation of specific keys in the computation parameter field. Metadata can only be read from or set on valid property data.

Valid field names are *bounds*, *comment*, *info*, *flags*, *parameters* and *unit*.

table move

```
table move tablehandle ?datasethandle|remotehandle? ?position?
t.move(?target=?,?position=?)
```

Make a table a member of a dataset, or remove it from a dataset. If the dataset handle parameter is omitted, or an empty string (or **`None`** for **PYTHON**), the object is removed from its current dataset. If it was not a dataset member, this command variant does nothing. The dataset handle may be the name of a remote dataset for moving objects over a network connection.

If a target dataset handle or reference is specified, the object is added to the dataset, if allowed by the acceptance bits of the dataset, and removed from any dataset it was member of before the execution of the command. By default the object is added to the end of the dataset object list, but the final optional parameter allows the specification of a dataset object list index. The first position is index zero. If the parameter value *end* is used, or the index is bigger than the current number of dataset objects minus one, the object is appended as per the default. It is legal to use this command for moving objects within the same dataset.

Another special position value is *random*. This value moves to the object to a random position in the dataset. Using this mode with remote datasets is currently not supported.

The dataset handle cannot be a transient dataset. It is not possible to move a table into its own built-in dataset.

The return value of the command is the dataset of the object prior to the move operation. It is either a dataset handle/reference, or an empty string (**TCL**) or **`None`** (**PYTHON**) if it was not member of a dataset

Examples:

```
table move $thandle $dhandle 0
table move $thandle
```

In the first sample line, the table is inserted as the first element in a dataset. The second line reverts this operation and removes the table from the dataset.

This command can be used with a remote dataset descriptor. In that case, the table is packed into a serialized object representation, transmitted over the network and restored as member of the remote dataset at the specified position. The local table is deleted if the transfer succeeds.

table movecolumns

```
table movecolumns tablehandle columnrange ?destination?
t.movecolumns(columnrange=,?destination=?)
```

Move one or more columns including their cell data within a table. If the destination, a simple column address, is not specified, the selected columns are moved to the right of the table. The destination cannot be in the range of the source columns.

The command may also be written as **`table movecol`** or **`table movecols`**.

The command returns the original table handle or reference.

table moverows

```
table moverows tablehandle rowrange ?destination?
t.moverows(rowrange=,?destination=?)
```

Move one or more rows, including their cell data within a table. If the destination, a simple row address, is not specified, the selected rows are moved to the bottom of the table. The destination cannot be in the range of the source rows.

The command may also be written as **`table moverow`**.

The command returns the original table handle or reference.

table mutex

```
table mutex tablehandle mode
t.mutex(mode)
```

Manipulate the object mutex. During the execution of a script command, the mutex of the major object(s) associated with the command are automatically locked and unlocked, so that the operation of the command is thread-safe. This applies to builds that support multi-threading, either by allowing multiple parallel script interpreters in separate threads or by supporting helper threads for the acceleration of command execution or background information processing. This command locks major objects for a period of time that exceeds a single command. A lock on the object can only be released from the same interpreter thread that set the lock. Any other threaded interpreters, or auxiliary threads, block until a mutex release command has been executed when accessing a locked command object. This command supports the following modes:

- *lock*
  Increase the recursive mutex lock count on the object. The command returns the current lock count after the command, excluding the transient single-command lock.
- *reset*
  Release all persistent locks on the object, if they exist.
- *test*
  Return the current persistent lock count on the object. This excludes the transient per-command lock.
- *unlock*
  Decrease the recursive lock count on the object. The command returns the current lock count after the command, excluding the transient single-command lock. Unlocking an object which has not been persistently locked results in an error.

There is no *trylock* command variant because the command already needs to be able to acquire a transient object mutex lock for its execution.

The command returns the current lock count.

table need

```
table need tablehandle propertylist ?mode? ?parameterdict?
t.need(property=,?mode=?,?parameters=?)
```

Standard command for the computation of property data, without immediate retrieval of results. This command is explained in more detail in the section about retrieving property data.

The return value is the original table handle or reference.

Example:

```
table need $th T_XYPLOT recalc
```

table new

```
table new tablehandle propertylist ?filterset? ?parameterdict?
t.new(property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **`table get`** command. The difference between **`table get`** and **`table new`** is that the latter forces the re-computation of the property data, regardless whether it is present and valid, or not.

table nget

```
table nget tablehandle propertylist ?filterset? ?parameterdict?
t.nget(property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **`table get`** command. The difference between **`table get`** and **`table nget`** is that the latter always returns numeric data, even if symbolic names for the values are available.

table nnew

```
table nnew tablehandle propertylist ?filterset? ?parameterdict?
t.nnew(property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading object data and attributes. It is explained in more detail in the section about retrieving property data.

For examples, see the **`table get`** command. The difference between **`table get`** and **`table nnew`** is that the latter always returns numeric data, even if symbolic names for the values are available, and that property data re-computation is enforced.

table normalize

```
table normalize tablehandle column ?singlerow?
t.normalize(column=,?singlerow=?)
```

Normalize a table column with numerical data to an average value of zero and a standard deviation of one.

If the data type of the column is a floating-point vector type (vectors of floats or doubles), and the *singlerow* flag is set, the normalization is performed individually on each vector in the cells of the selected column and not over all rows together.

The return value of the command is a list of the offset and scaling factors used to normalize the data column, in that order. In case the *singlerow* command variant was used, the return value is a list of these number pairs with one list element per row.

If the command is used on float or double vector columns in normal mode, all vector elements are treated as independent data values.

It is possible to normalize integer-type columns. However, since this command does not change the data types of the columns, only the offset and scaling values are reported but the column data is not changed because this cannot be done in a reasonable fashion.

Any attempt to normalize non-numeric columns results in an error.

table outerjoin

```
table outerjoin tablehandle1 tablehandle2 ?column1? ?column2? ?cmpflags?
t.outerjoin(table2=,?column1=?,?column2=?,?comparisonflags=?)
```

Perform a relational outer join on two tables and return the handle or reference of a newly created join table. The command arguments and result value are explained in the **`table innerjoin`** command.

The difference between a standard inner join and an outer join is that rows from either the first input table which do not match any values of the join column in the second table, or from the second input table which do not match any values of the join column in the first table, are still output, with all column data originating from the respective other table set to **`NULL`** values. The outer join is a fusion of a left and right join.

table pack

```
table pack tablehandle ?rowrangelist? ?columnrangelist? ?maxsize?
   ?compressionlib?
t.pack(?maxsize=?,?rows=?,?columns=?,?compressionlib=?)
```

Pack the table into a compressed, base64-encoded, serialized object string. This string can be used with a **`table unpack`** command to restore the table.

By default the full table is packed, but the optional range parameters, which are both range lists, can be used to pack a subset of the table.

The maximum size of the object string (default -1, meaning unlimited size) can be configured by the optional *maxsize* parameter. The size is specified in bytes. If the pack string would be longer than the maximum size, an error results.

Note that this command does not pack ensembles or reactions which are associated with a table. If a table is restored, this linkage is lost.

The default compression library is *zlib*. Other useful variants include *lzo* and *gzip* (and there are other internal types), but these may not be available on all builds due to license issues, and you need to specify the compression library when a dataset is unpacked. It is generally recommended to stay with *zlib*.

The command returns the pack string.

In **Python**, tables support the standard *pickle*/*unpickle* protocol.

Example:

```
set ts [table pack table0 {F Cl Br I} {vdwradius covradius}]
```

This command copies the Van der Waals and covalent radii of the halogens from the **PSE** table into a pack string. The string can be restored to a subset table with a

```
set thnew [table unpack $ts]
```

table poploop

```
table poploop tablehandle ?mode? varname ?objvarname? body
t.poploop(function=,?mode=?,?rowvariable=?,?objectsvariable=?)
```

Perform the **`table poprow`** command (see below) in a loop. The loop is repeated until no more rows can be extracted from the table. In each iteration, the extracted table data is stored in the specified variable, the row is deleted, and the code in the body executed. If execution of the body code results in an error, the loop is stopped immediately.

If an object variable name is specified, it contains a list or dictionary of the handles of the cell objects in the current row, with empty strings (**`None`** for **PYTHON**) as elements for cells which do not possess a cell object. These objects are unlinked from the current row and are no longer associated with the table. It may be necessary to dispose them in the loop if they are not linked to other controlling objects or moved to, for example, a target dataset.

In the **TCL** interface, the loop can be left early without raising an error by executing a standard **TCL** **`break`** or **`return`** loop control command in the body.

The **PYTHON** version of the loop method does intentionally have a different argument sequence for convenience. The function argument may either be a multi-line string (similar to the **TCL** construct), or a function reference. Functions are called with the table object reference, the row data tuple or dictionary and the row objects tuple or dictionary as three arguments.

Normal **PYTHON** functions have their own context frame, and are passed arguments, so that the specification of loop variables is not generally useful in that call style, though is is allowed. For string function blocks the code is executed in the local call frame, and the variable with the current object reference is visible locally. Script code blocks must be written with an initial indentation level of zero. Within the **PYTHON** functions, the normal *break* and *continue* loop control commands cannot be used to to scope limitations. Instead, the custom exceptions *BreakLoop* and *ContinueLoop* can be raised. These are automatically caught and processed in the loop body handler code.

There is no **PYTHON** iterator for this loop style.

The possible values of the optional *mode* argument are the same as in the **`table poprow`** command, as is the default mode *list*.

The command is thread-aware and will enter a wait loop until the **EOR** condition on the table has been met, so that other threads may fill the table.

The return value is the number of rows processed. The last value of the loop variables remain accessible outside the loop, if loop variables were used.

table poprow

```
table poprow tablehandle ?mode?
t.poprow(?mode=?)
```

Remove the first row of the table and return its content as result. The form of the result can be modified by the optional mode parameter. Its possible values are:

- *list*
  This is the default. The row data is returned as a standard **TCL** list.
- *dict*
  The row data is returned as a **TCL** dictionary. The primary column names are the dictionary keys. For columns without a name, a synthetic name in the form *col%d* is generated.

- *ens*
  Return an ensemble object. The source of the ensemble is, in this order, either an existing association of the row with an ensemble, the first column marked as structure source (see description of column flags), or an empty ensemble constructed on the fly. Column data which is not already present on the result structure is attached as ensemble properties if possible. If a column is already a property column, and the column property is an ensemble property, it is directly attached to the ensemble. For other column data, with the exception of explicit property data of incompatible objects, synthetic property definitions are automatically created. **`NULL`** data is skipped.
- *object*
  The return value is an automatically chosen major object, either an ensemble or a reaction, depending on the type of existing row associations, source columns, or present property types. This mode can also be selected as *auto* or *#auto*.
- *reaction*
  As above, but the result is a reaction object. The source is either an existing row association with a reaction, a reaction source column, or a newly constructed empty reaction.
- *rownamedict*
  The same as the *dict* mode, except that the dictionary also contains a *rowname* key with the name of the current row as value.
- *rownamelist*
  The same as the *list* mode, except that the first list element is the row name, not the value of the leftmost column.

For this command to work, the table needs to be editable. If there are no more table rows, an empty result is returned. If the table has an active background import thread, the command blocks until the import thread has added another row, or has terminated.

The command may be abbreviated as **`table pop`**.

table properties

```
table properties tablehandle ?pattern? ?noempty?
t.properties(?pattern=?,?noempty=?)
```

Return a list of the valid properties on the table object. If desired, the property list can be filtered by the optional string match pattern. Since tables incorporate no minor objects, only true table properties (standard prefix *T_*) are listed. Properties of ensembles or reactions associated with the table are not output because these are not component objects, just cross-references.

If the *noempty* flag is set, only properties where at least one data element is not the property default value are output. By default, the filter pattern is an empty string, and the *noempty* flag is not set.

The command may also be written as short form **`table props`**.

Example:

```
set plotproplist [table props $th T_*PLOT*]
```

table purge

```
table purge tablehandle propertylist/table ?emptyonly?
```

```
t.purge(?properties=?,?emptyonly=?)
```

Delete property data from the table. In contrast to performing property deletions on, for example, ensembles this operation does not branch out to properties which are stored on objects linked to the table. Ensemble or reaction references are not traversed because they are not true container object memberships.

This command only deletes proper table properties (usually starting with *T_*). If the object class name *table* is used instead of a property name, the data of all table properties is deleted.

The optional boolean flag *emptyonly* restricts the deletion to those properties where the value of a property is identical to the default.

table randomize

```
table randomize tablehandle ?seed?
t.randomize(?seed=?)
```

Shuffle the rows of the table in a pseudo-random fashion. If the seed argument, an integer, is provided, it is used to seed the random generator. If the command is called the first time without a seed, the current time stamp is automatically provided as seed and the random number sequence becomes unpredictable. Subsequent calls of the command without a seed parameter continue with the random number sequence defined by the seed, until the command is again called with a seed argument. The random generator is then re-seeded and the number sequence specific to the seed is re-started.

This command is useful for testing the robustness of algorithms with respect to data ordering.

The command returns the original table handle or reference.

table rank

```
table rank tablehandle ?method? ?{column ?direction ?cmpflags??}?..
t.rank(?method?,?(column,?direction,?cmpflags)???,...)
```

Rank the table rows according to data in specific columns. The command returns a list of the individual ranks of the rows, with one numeric list item per row and in order of the rows. It does not sort the table.

There are three different ranking schemes:

- *centered*
  If there are multiple rows of the same superiority, their rank is the average position in the result list. For example, if there are two entries which would occupy both second place, their rank is both 2.5, the average of position 2 and 3 in the list. The next rank assigned is fourth place.
- *olympic*
  If there are multiple rows of the same superiority, they all get assigned the highest unclaimed rank. The next tier receives ranks which skip those which would have been assigned had the ranks of the superior positions been all different. For example, if there are two rows with first rank, the next assigned rank is third. This is the default method.

- *sequence*
  If there are multiple rows of the same superiority, they all get assigned the highest open rank. The next tier get ranks in immediate sequence, without correcting for ties. For example, if there are two tied top rows, they both get first rank, and the next rank assigned is second.

The rest of the optional arguments select the columns to use for ranking, and the comparison operations used to determine the rank. Each argument is a list of one to three elements. The first list element, the column name, is mandatory. The special column name *#name* can be used to use the row name as criterion, *#row* or *#record* for the row number, and *#random* or *#rnd* as tiebreaker.

If the second element of a rank specification is not supplied, the sort direction for rank comparisons is *up*, meaning that lower values imply a higher (numerically lower) rank. The sort order can also be specified as *down*, which inverts this.

Finally, the comparison operation can be influenced by providing a list of comparison flags. These the same as used with the **`prop compare`** command.

In case multiple comparison columns are given, columns to the left have precedence over columns to the right.

Not specifying any comparison column is syntactically allowed, but reports all rows as of the same rank.

table reactions

```
table reactions tablehandle
t.reactions()
```

Return a list of the handles or references of all reactions which are referenced by the table. Every reaction is reported only once, even if it is referenced by multiple rows. Rows with reaction references are usually added to tables my means of the **`table addreaction`** command. In case a row has no reaction reference, it is ignored, and no output is produced.

table read

```
table read filename ?parameter value?...
table read filename ?dictionary?
Table.Read(filename,?parameter,value?,...)
Table.Read(filename,?dictionary?)
```

Read a table file into a newly allocated **CACTVS** table object and return the handle or reference of the new table if the operation succeeded. The column data types and other attributes of the new table are set appropriately according to the data found in the file. This is even true for pure **ASCII** text tables, because an attempt is made to promote the column formats to integers, floats or booleans after the data has been read initially as strings, if the data content allows the conversion.

Besides a normal file name, it is also possible to specify *stdin* as magic file name to read from standard input, or a pipe by starting the file name argument with the "|" character. If a normal file is used, and the suffix cannot be identified by the currently loaded table file format I/O handlers, an attempt is made to auto-load a handler by constructing a standard module name from the suffix (see **`tablex`** command for more information about table file format handlers). Additionally, some **URL** formats, such as **`http(s)://`**, **`(s)ftp://`** and **`gs://`** (**GOOGLE** storage) are also recognized as file names. For these formats, the file is automatically temporarily downloaded, read, and again deleted from local disk space. Access credentials can be embedded in the **URL** in standard fashion.

For normal disk files, but not when reading from *stdin*, pipes or **URL**s, the file format is automatically detected by looking into the file content, independently of the file suffix. This is done by invoking in reverse load order the detection function of all currently loaded table I/O modules and the built-in handlers. For build-in table handlers, but only for those, such as the simple separator-controlled text table dumps, the table file may be **`gzip`**-compressed (but not **`bz2`**, etc.), and the format is still auto-detected. For other table disk files, the presence of **`gzip`** or **`bz2`** compression is detected, but no automatic format detection takes place behind the compression layer. These files can still be read without prior decompression by explicitly stating the format in the argument dictionary.

Additional optional parameters to control the input can be specified either as a number of keyword/value argument pairs, or a single dictionary argument. The recognized control keywords are the same for both variants.

- *colnames*
  In some cases it may not be possible to automatically detect whether the table file contains column names or not in the first data line. The boolean *colnames* parameter allows the programmer to provide this information explicitly. If the *title* reader hint is also used, the column names are expected to come after the title line.
- *debug*
  A boolean flag indicating that, if set, debug information should be printed while reading the file. This is primarily a developer option.
- *firstrecord*
  The first row record (starting with one) to read. By default, table input starts at the first row. Not all table file format I/O modules support this.
- *format*
  By default, the file format is identified automatically. However, in order to enforce reading the input source as a specific format, for example in case of an ASCII table file with confusing separator characters, it is possible to explicitly state the format with the *format* parameter. If the handler for that format is not yet loaded, an attempt at auto-loading it is made.
- *headeronly*
  A set boolean parameter instructs the table I/O format handler to only read the table column definitions and table-level properties. This is not supported on all table formats. If a table I/O module does not support this input mode, the full table is read instead, as by the default operation. This includes all the information which would be read in header-only mode, but also all row and cell data, which requires more memory and takes longer to read. If header-only mode is supported, the result table has no rows and no cells after input, and also no associated row chemistry objects.
- *maxrows*
  If set to a non-negative integer value, this parameter limits the maximum number of rows to read from the table. By default, the maximum row count is not limited.
- *rownames*
  In some cases it may not be possible to automatically detect whether the table file contains row names or not. The boolean *rownames* parameter allows the programmer to provide this information explicitly.

- *scoped*
  If set, directly set the scoped attribute, which makes the table invisible from TCL interpreters other than the one performing the read.
- *separator*
  This parameter provides a way to explicitly define a separator character for ASCII-style text tables. By default, an attempt is made to determine the separator character automatically if the table file format is identified as an ASCII-based format with a variable separator. This option is ignored for file formats with a more advanced internal structure.
- *sheetname*
  CACTVS table objects can represent only a single table, not multiple tables forming a sheet collection, notebook or similar construct in the input file. In case files are read which contain multiple tables, for example *sqlite, xlsx* or *fits* files, the desired table can be selected via the *sheetname* string option. The alternative parameter name *table* is an alias for *sheetname*. If a multi-table file is read, and no such sheet name provided, the first table from the file is the one extracted. There is currently no way to read multiple tables from such a file with a single command, nor a method to query the number or names of sheets or tables present.
- *skiperrors*
  If this boolean parameter is set, rows which cannot be decoded are ignored. This is useful for example in case there are trailing blank or comment lines in the file. However, the read command will then also remain silent in case *all* rows fail, for example because of a mis-identification of the file format, so this option should be used with care.
- *table*
  An alias for *sheetname*, for a more familiar naming if, for example, a specific table from a database file is read.
- *target*
  If set to a table handle, the data is read into this existing table, which is fully reset before input commences. By default, a new table object is allocated.
- *title*
  A boolean flag indicating whether the table reader should interpret the first line as title string. If column names are also read, these follow the title line. The title string is stored in the *title* table object attribute.

If a read table file contains, besides the cell data, structure or reaction object data, as it is possible for example with native CACTVS or KNIME table files, these objects are automatically read and moved into the internal table dataset object. Proper row references to these objects are also established.

`table load` is an alternative command name.

Example:

```
set th [table read $file colnames 1 separator “!”]
```

This reads a slightly unusual ASCII table with exclamation marks as separators, and column names in the first row.

table readblob

```
table readblob data ?parameter value?...
table readblob data ?dictionary?
Table.Readblob(data,?parameter,value?,...)
Table.Readblob(data,?dictionary?)
```

This command is the same as the **`table read`** command, except that is reads from an in-memory blob instead from a file. If the responsible table I/O module for the table file format does not support in-memory decoding, a temporary file is automatically created and removed after reading.

Currently, automatic format detection of in-memory table data does not work in some cases, or forces the inefficient use of a temporary file. It is advisable to specify an explicit file format in the optional arguments to avoid this problem.

`table loadblob` is an alternative command name.

table recalc

```
table recalc tablehandle ?rowrangelist? ?columnrangelist?
t.recalc(?rows=?,?columns=?)
```

Recompute the values of all cells which are linked to column computation functions in the specified block. The default re-computation block is the complete set of rows and columns. This command does not affect cell data which is not linked to a function.

The return value is a boolean status code indicating whether all requested values could be computed.

table ref

```
Table.Ref(identifier)
```

PYTHON only method to get a table reference from a handle or another identifier. For tables, other recognized identifiers are table references, integers encoding the numeric part of the handle string, the **`UUID`** of the network object, or its name.

table rename

```
table rename tablehandle srcproperty dstproperty
t.rename(srcproperty=,dstproperty=)
```

This is a variant of the **`table assign`** command. Please refer the command description in that paragraph.

table replot

```
table replot tablehandle ?rowrange? ?columnrange?
t.replot(?rows=?,?columns=?)
```

Replot structure and reaction images embedded in the table. The default replot block is the complete set of rows and columns. The command only affects data cells of columns of type *image*.

The command returns the original table handle or reference.

table rewind

```
table rewind tablehandle
t.rewind()
```

Reset the table iterator record. This is equivalent to setting the *record* table attribute to one.

The command returns the original table handle or reference.

table rightjoin

```
table rightjoin tablehandle1 tablehandle2 ?column1? ?column2? ?cmpflags?
t.righjoin(table2=,?column1=?,?column2=?,?comparisonflags=?)
```

Perform a relational right join on two tables and return the handle or reference of a newly created join table. The command arguments and result value are explained in the **`table innerjoin`** command.

The difference between a standard inner join and a right join is that rows from the second input table which do not match any values of the join column in the first table are still output, with all column data originating from the first table set to **`NULL`** values.

table rownamedictloop

```
table rownamedictloop tablehandle rowvariable ?rowobjectsvariable? ?maxrows?
   ?offset? body
t.rownamedictloop(function=,?maxrows=?,?offset=?,?rowvariable=?,
   ?objectsvariable=?)
```

This command is a variant of the **`table loop`** command. It stores the row data as a dictionary in the loop variable, with the column names as keys. In addition, the dictionary key *rowname* holds the name of the current table row. The value of the iterator style table attribute is ignored.

Please refer to the **`table loop`** command description for more information.

table rownamelistloop

```
table rownamelistloop tablehandle rowvariable ?rowobjectsvariable? ?maxrows?
   ?offset? body
t.rownamelistloop(function=,?maxrows=?,?offset=?,?rowvariable=?,
   ?objectsvariable=?)
```

This command is a variant of the **`table loop`** command. It stores the row data as a simple list in the loop variable. The first list element is the name of the current row. The value of the iterator style table attribute is ignored.

Please refer to the **`table loop`** command description for more information.

For the sake of compatibility with the **Python** interface syntax, the command may also be invoked as **`table rownametupleloop`**.

table scan

```
table scan tablehandle/queryhandle expression ?mode? ?parameterdict?
t.scan(query=,?mode=?,?parameters=?)
```

This command is a variation of the **`dataset scan`** or **`molfile scan`** commands. The query expression is matched, in row order, on all ensemble or reaction objects associated with the table rows, i.e. the same objects as reported by the **`table ens`** and **`table reactions`** commands. An attempt is made to automatically instantiate these objects if a row currently has no association, but a structure source column has been configured via the *structuresource* or *reactionsource* attribute. Rows which do not possess a structure or reaction reference even after this attempts are skipped and can never return a match.

The default retrieval mode for this scan command variant is *rowlist*, which is an alias of the *indexlist* mode of dataset scans. In this mode, matching row indices are returned as a list. If an ensemble or reaction is associated with multiple rows, it is tested repeatedly for every row it is associated with.

For further information on the operation of this command and the use of the various arguments, refer to the paragraph on **`dataset scan`**.

The optional parameter dictionary is the same as for **`molfile scan`**, but not all parameters are actually used. At this time, only the *matchcallback, maxhits, maxscan, order, progresscallback, progresscallbackfrequency, sscheckcallback, startposition* and *target* parameters have an effect. In case a progress callback function is used, the table handle is passed as argument in place of the *molfile* handle in **`molfile scan`**.

This command does not operate on table cell data, but exclusively on the associated structures, reactions, and their present or computable data. For queries on table cell data, use the **`table find`** command. However, the *delete* query mode of this command does indeed remove matching table rows, not delete the associated objects.

table select

```
table select tablehandle column|all operator value ?mode? ?retrievalcolumn?
t.select(column=,operator=,value=,?mode=?,?retrievalcolumn=?)
```

Perform a scan over the table and mark all rows which match a value. This command processes multiple matches. For a simple version of the command which stops after the first match, see **`table find.`**

The default mode is *new*, which resets all current *select* flags on the rows before running the command. The alternative modes are *or*, which just sets additional flags, *and*, which resets the selection flag for rows which are currently selected but not pass the new expression, and *eor* or *xor*, which invert the selection flag on all rows which are matched.

The column argument is either the name or index of an existing table column, or the magic word *all*, which runs the scan on all columns. There is currently no support for scans which use more than one, but not all columns. The special column name *#name* to test the row names is also supported.

For single-column scans, the value argument needs to be a string which can be decoded as the data type of the scanned column. For *all* scans, the argument is parsed separately for each column. Only columns where the data successfully decodes according to their data type are searched. Other columns where the value argument cannot be parsed are silently skipped.

The operator argument is a standard arithmetic operator, optionally modified by either single-character modifier flags, or specified as a list with the symbolic names of these modifier flags. The syntax of this argument is compatible to that of property query leaf expressions of the **`molfile scan`** command. The operators *in* and *notin* are also supported. In that case, the value argument must be a valid list. The list is split, and each element parsed and matched separately. For the *in* operator, rows where any of the multiple elements match with an equality comparison are selected. For a *notin* match, none of the elements must match in an equality comparison.

Examples:

```
table select $thandle E_NAME *= "*chloro*"
table select $thandle E_NAME {= shell} "*chloro*"
table select $thandle E_WEIGHT <= 200
table select $thandle SID in [ens get $eh E_SIDSET]
```

The first two sample statements are equivalent and both perform a shell-syntax string match on the column `E_NAME`, matching any data which contains the substring *chloro*. The third example performs a simple numeric comparison, and the last example tests whether the *SID* column data value is one of a list of alternatives.

The result is by default a list of the row indices of all selected rows after the operation. In addition, the selection attribute on matched rows is set and can be queried by attribute retrieval commands. If the name of a retrieval column is set, the return value is a list of the cell data of matching rows of that column. The special column name *#name* for access to row names is also supported. The manipulation of the row selection flag is not changed if this retrieval type is used instead of the default index retrieval.

This command has limited power in the types of comparison operations it supports. For a more extensive search capability please refer to the **`table sqlselect`** command. However, in many cases this simpler command is much faster than the **SQL** version, especially if it operates on columns with indices and uses a search operator which can make use of the index information.

This command does not support data access via linked ensembles or reactions. It can only test cell data directly stored in the table.

table set

```
table set tablehandle ?property/attribute value?...
table set tablehandle ?dictionary?
t.set(property/attribute,value,...)
t.set({property/attribute:value,...})
t.property/attribute = value
t[property/attribute] = value
```

Standard data manipulation command. It is explained in more detail in the section about setting property data.

Examples:

```
table set $thandle T_COMMENT "This data, if leaked in an appropriate fashion, will
lead to the waste of years of research by our competitors"
```

The command can also be used to set a rich set of table attributes. The list of attributes is documented in the section on the **`table get`** command.

Example:

```
table set $thandle bgcolor "grey80" imagedirectory "/www/images" imageurl "."
```

table setcell

```
table setcell tablehandle row column ?attribute value?...
table setcell tablehandle row column value
t.setcell(row,column,?attribute,value?,...)
t.setcell(row,column,value)
t.setcell(row,column,{attribute:value,...})
t.setcell(row,column,(attribute,value,...))
```

Set the cell data value, or a cell attribute. The second form is a shortcut for setting the cell data value and equivalent to using an explicit *value* attribute name. In the **PYTHON** variant, the attribute set can also be specified as a dictionary.

The following cell-level attributes are supported:

- *bgcolor*
  The background cell color, for output in formats which support this.
- *class*
  The class attribute of `<td>` tags when writing html output.
- *comment*
  A free-form text comment.
- *fgcolor*
  The foreground cell color, for output in formats which support this
- *font*
  The name of a font to use in rendering. This overrides fonts set on the cell column or top-level table object.
- fontsize
  The size of the font to use in rendering. This overrides font sizes set on the cell column or top-level table object.
- *format*
  Cell-level data format flags. This is the same set as for the global table attribute of the same name, which is explained in the section describing the `table get` command.
- *image*
  A **Tk** image handle if the table object is linked to a **Tk** table widget.
- *linktext*
  A text overlaying a link to use when formatting **URL**s in formats like **HTML**. An empty string results in the link value to be printed verbatim.
- *note*
  A cell note, which can be a block of arbitrary bytes, not just a string. In output formats such as **MS Excel** this is displayed as a tool tip.
- *object*
  A per-cell chemistry object. Currently, these must be ensembles or reactions (specified by their handles), or an empty string in order to remove an existing cell object. A cell object has precedence when computing cell data over a row object, if one is simultaneously present. An ensemble or reaction cell object cannot be deleted by `ens/reaction delete` commands. It is deleted by another `table setcell` command, or when the row, column or complete table is destroyed. Cell objects are stored in retrieved in the native **Cactvs** table format, even in output modes where row objects are not. Other table file formats do not preserve these. Setting a cell object duplicates the source object, just like setting an ensemble or reaction property value.

- *objectref*
  This is the same operation as the *object* command, except that the cell object is not copied. At the time the command is executed, the new cell object must be deletable (i.e. not a property value, and not a cell object anywhere else). After it has been set, is is managed by the table and cannot be deleted by normal means. When the table cell is deleted, or another cell object is set, this object is destroyed.
- *shape*
  Set a **KNIME**-compatible graph rendering shape for the cell. Valid shape names are *default, rectangle, circle, triangle, revtriangle, diamond, asterisk, cross, xshape, hline* and *vline*.
- *shapesize*
  Set a **KNIME**-compatible graph rendering shape size multiplier for the cell. This is a floating-point attribute, and 1.0 is the default.
- *tag*
  The cell tag in a linked **Tk** table widget.
- *value*
  The cell data value. It must be encoded in a format which can be decoded as the data type specified in the column definition.
- *window*
  The name of a **Tk** window displaying cell contents.

The attributes *image*, *tag* and *window* are only useful for developers who want to display a **Cactvs** table in a **GUI** tool developed with the **Tk** toolkit and its table widget.

The command returns the original table handle or reference.

table setcolumn

```
table setcolumn tablehandle column ?attribute value?...
t.setcolumn(column,?attribute,value?,...)
t.setcolumn(column,{attribute:value})
t.setcolumn(column,(attribute,value,...))
```

Set column attributes. The following column-level attributes can be set:

- *alias*
  An alias name for the column which can be used to identify it in addition to its numerical index or primary column name.
- *bgcolor*
  The background column color, for output in formats which support this.
- *bgcolorfunction*
  An **SQL**-style function to compute the background color. If this is set, it has precedence over direct color specification via the *bgcolor* attribute, but has lower precedence that a background color specified directly on a cell.

- *celldata*
  Set the cell values of the column starting with the first row. The attribute argument is expected to be a list, which is split into per-cell values. If there are fewer list elements than rows, the remaining rows of the current column are not touched. If there are more list elements than rows, additional rows are added, which have **NULL** values in all columns except the current one. *data*, *value* or *cellvalue* are alias names for this attribute. To set all cells of the column to the same value, use the *coldata* attribute.

- *class*
  The class attribute of table cells in **HTML** output, if not overridden by an explicit table cell class.

- *coldata*
  Set the cell values of the column to a common value. Use the *celldata* attribute to set column cells to different values via a value list. *colvalue* is an alias attribute name.

- *comment*
  A free-form text comment.

- *dataformat*
  A formatting string which is used in text-oriented formats (i.e. **HTML**, **EXCEL**) to reformat the data cell value on file output. Every instance of *'%s'* in the formatting string is replaced by the cell value string, and that result be written to the file. The original cell content remains unchanged. With an empty formatting string (the default) the normal raw data formatting is used.

- *datatype*
  The column data type, which can be any data type a property definition can use. If there are already rows in the table when the command is executed, an attempt is made to cast the column values from their original data type to the new one. If this does not succeed, failing cells are set to **NULL** values. If the column is associated with a property, and the new data type is not storage-compatible with the native property datatype, the column connection with the property is lost.

- *dbdefault*
  This attribute is only used in SQL output. If it is not empty, its value is used to specify a column default value as per the SQL standard in the written table definition. The value is used verbatim in output, so depending on the column type, it may have to be properly quoted in SQL fashion.

  The special value *auto* or *#auto* can only be used when there are already data rows in the table. With this command argument, and if the current data type is *string*, and the columns are not associated with a property and its native data type, the string contents of the data cells are analyzed across all rows, an optimally matching numerical data type (*boolean*, *color*, *integer*, *double*, *date*, *uint64*, *intvector*, *doublevector*) is determined and the column data type as well as the cell data contents are updated. This variant is usually used after reading some table data from a file in a format without explicit column data type information. For these inputs, the data is initially stored as string columns in the table object.

- *dbflags*
  This attribute encodes a set of hint flags for setting up **SQL** database columns for storing data of this table column. It format is the same of the *dbflags* attribute of property definitions (see **`prop set`** command). This column flag overrides the corresponding flag of a property linked to the column.
- *dbforeignkey*
  This attribute is only used in SQL output. If it is not empty, its value is used to specify a foreign key relationship as per the SQL standard in the written table definition. The format is *foreigntable(foreigncolumn)*. The value is used verbatim in output, so if the foreign table or column contains unusual characters or whitespace, they must be properly quoted in SQL fashion. The allowed syntax of this attribute in the context of more exotic table or column names is dependent on the target SQL dialect and not checked when the attribute is set.
- *dbindex*
  Specify the type of index to be used on **SQL** database tables which are derived from this **CACTVS** table column. Possible values are *none* (the default, no index), *generic* (simple index with support for multiple identical values) and *unique* (primary index with no duplicate column data values).
- *description*
  A free-form textual description of the column.
- *displaywidth*
  A field output formatting width, usually defined in characters. This attribute is different from the *fieldlength* attribute, which defines an inherent data size. If the width is negative, the attribute is considered unset.
- *editable*
  A boolean flag indicating whether this column is editable or not. The default is editable.
- *fgcolor*
  The foreground column color, for output in formats which support this.
- *fgcolorfunction*
  A **SQL**-style function to compute the foreground color. If this is set, it has precedence over direct color specification via the *fgcolor* attribute, but has lower precedence that a foreground color specified directly on a cell.
- *fieldlength*
  The inherent field length, in characters or significant digits. This is attribute is different from the *displaywidth* attribute, which is only used for output formatting. A negative value defines the attribute as unset.
- *flattening*
  A boolean flag indicating if the column data should automatically be temporarily flattened (see **`table flatten`** command) on output.
- *font*
  The column-level output font name, or an empty string if it is not set.
- *fontsize*
  The column-level font size in points. If not set, zero is reported.

- *format*
  Column-level data format flags. This is the same set as for the global table attribute of the same name, which is explained in the section describing the **`table get`** command.
- *function*
  A **SQL**-style generator expression for the data in the column cells. This attribute has an effect only if the column type if type *function*. Cell data values are automatically recomputed when the function definition changes, or any of the data in the column the function references.
- *headerbgcolor*
  The background color of column headers. If not set, use an empty string.
- *headerdata*
  A data value for the column header data field. When set, it must be decodable by the input function for the column header data type. Note that this is data for the independent header data field, not for the main column cells.
- *headerfgcolor*
  The foreground color of column headers. If not set, use an empty string.
- *headerfont*
  The font to use for column headers, or an empty string if not set.
- *headerfontsize*
  The size of the font for column headers. If not set, zero is reported.
- *headerformat*
  Column header data format flags. This is the same set as for the global table attribute *format*, which is explained in the section describing the **`table get`** command.
- *headerproperty*
  The name of a property to associate the header data field with. If it is changed, existing column header data is discarded. Note that this is property which is independent of the main column cell property (attribute *property*).
- *imageparameters*
  A keyword/value parameter set to be used for the computation of images. This attribute has an effect only on columns of type *image*. Any parameter defined here overrides the parameter of the same name in the global parameter settings on the image property (i.e. `E_GIF`, `E_EMF_IMAGE`, etc.). The parameter override is only temporary while the image data is computed. The global parameter set is not changed permanently.
- *imageborder*
  An integer image border value to be used when images are embedded in some output formats. For example, in **HTML** output this is used for the *border* attribute in the <*img*> tag.
- *indexed*
  A boolean flag indicating whether the data of the table column is currently indexed or not. Index creation or re-creation can be forced by setting it to *true*, and the index can be torn down by setting it to *false*. Note that this is not the same as the database index attribute *dbindex*. This index is an in-memory index on the column data which is used to accelerate simple look-ups via the **`table find`** command group. The read-only attribute *index* (see **`table getcol)`** is again not the attribute as this one.

- *isname*
  A boolean flag indicating that the column data contains names of persons. If set, table sort and comparison functions use a special string comparison algorithm to deal with titles and other personal name prefixes properly. This flag is ignored for columns which do not hold string data.
- *linktext*
  A text overlaying a link to use when formatting **URL**s in formats like **HTML**. An empty string results in the link value to be printed verbatim.
- *mimetype*
  An indication of the format of data held in blob or disk files. If the table column is linked to a property, the value of the property is copied when the column is created. This attribute is used for example when mapping blob columns to different KNIME image column types.
- *name*
  The primary name of the column. Columns should avoid duplicate names, though this is not illegal. In case multiple identical column names are desired, a suitable set of unique and different *alias* names should be configured, and columns only be identified by these alias names.
- *note*
  An arbitrary sequence of bytes attached as a column-level note.
- *precision*
  The precision of numerical column data values, in significant digits. This attribute overrides any precision defined for properties linked to the column. Negative precisions (presumably meaning that the numeric value is only precise to tens, hundreds etc. range) are currently not supported. Setting this attribute also sets the *precision* column format flag as a side effect.
- *property*
  The name of the property the column data is associated with. If this attribute is set, it also automatically defines the column data type, precision and unit by copying those values from the property definition record. If the property is changed, any current data values on this column are deleted. If the table rows are still associated with chemical objects, an attempt is made to re-fill the column data cells.

  It is possible to define table columns which only refer to a specific field of a property. In that case, the name of the property is specified with standard field access syntax:

  ```
  table setcol $thandle $col property E_IRSPECTRUM(source)
  ```
- *reactionproperty*
  A boolean flag indicating that the data in this table column is a reaction identifier, but not a direct structure encoding like a **Reaction SMILES** string or an **RXN** blob. This information can be used to restore structure information for tables where the rows have no references to external ensemble objects.
- *reactionsource*
  A boolean flag indicating that the data in this table column is a string-based reaction encoding in a format which could, for example, be decoded with a **`reaction create`** command. This information can be used to restore reaction information for tables where the rows have no references to external reaction objects.

- *resetcellformat*
  This is a pseudo attribute which can only be set. It takes a boolean argument. If it is true, all cell-level cell formats, which override the column/row/table-level formatting, are reset for this column.

- *shape*
  Set a **KNIME**-compatible graph rendering shape for the column. Valid shape names are *default, rectangle, circle, triangle, revtriangle, diamond, asterisk, cross, xshape, hline* and *vline*.

- *shapesize*
  Set a **KNIME**-compatible graph-rendering shape size multiplier for the column. This is a floating-point attribute, and 1.0 is the default.

- *structureproperty*
  A boolean flag indicating that the data in this table column is a structure identifier, but not a direct structure encoding like a **SMILES** string or an **SD** blob. This information can be used to restore structure information for tables where the rows have no references to external ensemble objects.

- *structuresource*
  A boolean flag indicating that the data in this table column is a string-based structure encoding in a format which could, for example, be decoded with an `ens create` command. This information can be used to restore structure information for tables where the rows have no references to external ensemble objects.

- *transform*
  A transformation function for numerical table data values. The currently supported types are *linear* (the default), *inv* (negated), *abs* (use absolute value), *log* or *ln* (natural logarithm), *log10* (decadic logarithm), *log2* (logarithm base 2), *-log* (negative natural logarithm), *-log10* or *p* (negative decadic logarithm), *-log2* (negative logarithm base 2), *percent* (multiplied by 100) and *score* (divided by 100). This attribute is for documentation only. The column data values are not updated by changing this attribute.

- *type*
  The table column type. Possible values are *none*, *data* (data defined via a property or at least a data type), *image* (an image of the structure or reaction associated with a data row, with the image property automatically chosen to be suitable for a specific output format) and *function* for data columns which are dynamically computed and updated from the contents of other columns by means of **SQL**-style expressions (see *function* attribute). If the column type is changed, any previous cell data for that column is deleted. If the table rows are still associated with chemical objects, and attempt is made to re-fill the column cells.

- *unit*
  A free-form string describing the unit of the data.

- *urn*
  A **URN** associated with the column.

- *vectorsize*
  If the data type of the column is a vector type, set the size of the individual cell data vectors in the column to a specified common size. If vectors get enlarged beyond their original size, they add 0 or **`NULL`** element values. If the size is smaller than that of an existing vector, it is truncated. If the column data type is not a vector, the command is ignored.

The command returns the original table handle or reference.

`table setcol` and **`table colset`** are command aliases.

table setparam

```
table setparam tablehandle property ?keyword value?...
table setparam tablehandle property dictionary
t.setparam(property,?key,value?...)
t.setparam(property,dict)
```

Set parameter values in the metadata section of existing property data attached to the table. This command does not change the parameters for computations in the property definition (see **`prop setparam`** command for this function). It only stores its data in the parameter set which was copied into the metadata when the property was computed for the table.

This command does not attempt to compute property data. If the specified property is not present, an error results.

table setrow

```
table setrow tablehandle rowrange ?attribute value?...
t.setrow(rowrange,?attribute,value?,...)
t.setrow(rowrange,{attribute:value,...})
t.setrow(rowrange,(attribute,value,...))
```

Set row attributes for one or more table rows. The following row-level attributes can be set:

- *alias*
  An alias name for the row which can be used to identify it in addition to its numerical index or primary row name.
- *anchor*
  A string as navigation aid pointing to this row. This attribute is only used in **HTML** output, where it is output as a **`<a name="..."> `** tag.
- *bgcolor*
  The background row color, for output in formats which support this.
- *class*
  The class attribute of the **`<tr>`** tag. Only used in html output.
- *data*
  A list of the cell values for the row. The length of the list must be the same as the number of table columns. If more than one row is selected, the same values are entered for all these rows. *values* is an alias for this attribute.
- *editable*
  A boolean flag indicating whether this row is editable or not. The default is editable.

- *ens*
  Associate an ensemble with the table row. If this ensemble is not the same as the one currently associated, the old row data is invalidated, and an attempt is made to re-fill the row with data from the new ensemble. If the argument is an empty string, the row is explicitly set to be not associated with an ensemble. On retrieval, if no explicit or null ensemble association is set, but a column of the table is marked as structure source (see `table setcol` command), an attempt is made to create the ensemble from the data in the columns marked with this flag. These are scanned from left to right. If the data conversion from any of those succeeds, the new ensemble is automatically and permanently associated with the row, and the scan is stopped. An automatically created ensemble exists independently of the table in the normal chemical objects set, and must be explicitly deleted when no longer needed. *structure* is an alias for this attribute.
- *fgcolor*
  The foreground row color, for output in formats which support this.
- *font*
  The row-level output font name, or an empty string if it is not set.
- *fontsize*
  The row-level font size in points. Setting is to zero disables the attribute for this row.
- *format*
  Row-level data format flags. This is the same set as for the global table attribute of the same name, which is explained in the section describing the `table get` command.
- *height*
  An integer attribute to set the row height in various output formats, for example **Excel** or **HTML**. The unit depends on the output formats - usually these are pixels.
- *label*
  Associate the row with a minor object label. This makes only sense if the table row is also associated with an ensemble, and holds minor object data, such as atom or bond properties. If the value is set to minus one, the label is undefined.
- *linktext*
  A text overlaying a link to use when formatting **URL**s in formats like **HTML**. An empty string results in the link value to be printed verbatim.
- *name*
  Set the name of the row as arbitrarily formatted string. By default, and if row names were not read from file, rows do not have names and are addressed by their numerical indices. If an empty string is passed, this is interpreted as no name. It is not illegal to have multiple rows with the same name, but in that case care should be taken to identify them by their row index, or a suitably set unique alias.
- *note*
  An arbitrary sequence of bytes attached as a row-level note.
- *reaction*
  Associate a reaction with the table row. If this reaction is not the same as the one currently associated, the old row data is invalidated, and an attempt is made to re-fill the row with data from the new reaction. If the argument is an empty string, the row is explicitly set to be not associated with a reaction. On retrieval, if no explicit reaction or null association is set, but a column of the table is marked as reaction source (see `table setcol` command), an attempt

is made to create the reaction from the data in the columns marked with this flag. These are scanned from left to right. If the data conversion from any of those succeeds, the new reaction is automatically and permanently associated with the row, and the scan is stopped. An automatically created reaction exists independently of the table in the normal chemical objects set, and must be explicitly deleted when no longer needed.

- *shape*
  Set a **KNIME**-compatible graph rendering shape for the row. Valid shape names are *default, rectangle, circle, triangle, revtriangle, diamond, asterisk, cross, xshape, hline* and *vline*.
- *shapesize*
  Set a **KNIME**-compatible graph rendering shape size multiplier for the row. This is a floating-point attribute, and 1.0 is the default.
- *selected*
  Mark the rows as selected or deselected, depending on the value argument.

The command returns the original table handle or reference.

`table rowset` is an alias.

table show

```
table show tablehandle propertylist ?filterset? ?parameterdict?
t.show(property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the `table get` command. The difference between `table get` and `table show` is that the latter does not attempt computation of property data, but raises an error if the data is not present and valid. For data already present, `table get` and `table show` are equivalent.

table sort

```
table sort tablehandle ?{column ?direction ?cmpflags ?cmpvalue???}?...
t.sort(?(column,?direction,?cmpflags,?cmpvalue)????,...)
```

Sort the rows of a table according to the cell values in one or more columns. Every sort criterion argument after the table handle argument is a list or tuple of between one and four elements. In the simplest case, it is just the name of a single column. Additionally, a sort direction (*up* or *down*, the default is *up*) can be specified, plus comparison flags. The comparison flags argument supports the same set of flags as in the `prop compare` command and is explained there in detail.

If a comparison value is supplied as fourth argument, the sort utilizes the comparison results of cell values against this value for ranking, not the direct comparison result between the cell values. This is for example useful when sorting according to a bitvector similarity value to an external structure.

Sort columns to the left in the column specification list have precedence over those on the right. The special column name *#name or #rowname* can be used to sort on the row names, *#row* (or *#record*) to sort on the row number, and *#random* to sort on a random value assigned to every row, effectively duplicating the `table randomize` command. The row number criterion is also always added implicitly as an additional rightmost sort column to yield a stable sort. If an unstable sort is required, add a final explicit *#random* sort column, which has precedence over the implicit row number sort.

The command returns the original table handle or reference.

table split

```
table split tablehandle row
t.split(row)
```

Split a table into two. The original table retains all data rows up to, but not including, the split row. A new table with the same basic attributes, properties and column layout is created and gets the rest of the original data.

The command returns the handle or reference of the new table.

table splitcolumn

```
table splitcolumn tablehandle column ?separators? ?separatormerge?
t.splicolumn(column=,?separators=?,?separatormerge=?)
```

Split a table column into multiple result columns. This also changes the table column data type to *string*, which can have indirect consequences (see **`table setcol .. datatype`**).

The default split characters are whitespace. If the separator argument is set, every character in the separator string is an equivalent split character. It is also possible to set the separator to an empty string, which performs splits of cell data into individual characters. By default, every encountered separator character defines a new split position. The the merge flag is set to *true*, every *sequence* of separator characters is a split position instead. For example, a series multiple white spaces in the input then defines only a single word split, not multiple splits producing some empty result words.

The original split column retains the first result word after the split. Additional columns are inserted to the right of the source column, with the cell which contains the largest number of words defining the total number of columns added. **`NULL`** cells are not split, and result cells in added columns for which no split word was found because there are less words in the source cell than in the cells with the largest number of words, or the source cell is **`NULL`**, are also set to **`NULL`**.

The return value is the number of columns added, which is one less than the largest number of words found in a cell, if there were any non-**`NULL`** cells.

`table splitcol` and **`table colsplit`** are command aliases.

table sqldget

```
table sqldget tablehandle propertylist ?filterset? ?parameterdict?
t.sqldget(property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **`table get`** command. The differences between **`table get`** and **`table sqldget`** are that the latter does not attempt computation of property data, but initializes the property value to the default and returns that default, if the data is not present and valid; and that the **SQL** command variant formats the data as **SQL** values rather than for **TCL** or **PYTHON** script processing.

table sqlfind

```
table sqlfind tablehandle query ?maxrows? ?offset? ?mode?
t.sqlfind(query=,?nmax=?,?offset=?,?mode=?)
```

This command is very similar to the **`table sqlselect`** command, except that the query execution stops after the first matched row.

The result is the index of the matched row, or minus one in case no row matches.

table sqlget

```
table sqlget tablehandle propertylist ?filterset? ?parameterdict?
t.sqlget(property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **`table get`** command. The difference between **`table get`** and **`table sqlget`** is that the **SQL** command variant formats the data as **SQL** values rather than for **TCL** or **PYTHON** script processing.

table sqlnew

```
table sqlnew tablehandle propertylist ?filterset? ?parameterdict?
t.sqlnew(property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **`table get`** command. The differences between **`table get`** and **`table sqlnew`** are that the latter forces re-computation of the property data, and that the **SQL** command variant formats the data as **SQL** values rather than for **TCL** or **PYTHON** script processing.

table sqlselect

```
table sqlselect tablehandle query ?maxrows? ?offset? ?mode?
t.sqlselect(query,?nmax=?,?offset=?,?mode=?)
```

Perform a **SQL**-style *select* operation on the table and set the *selected* row flags depending on whether the row was selected by the query. This command yields multiple matches. For a simple version of the command which stops after the first match, see **`table sqlfind`**.

The default mode is *new*, which resets all current *select* flags on the rows before running the command. The alternative modes are *or*, which just sets additional flags, *and*, which resets the selection flag for rows which are currently selected but not pass the new expression, and *eor* or *xor*, which invert the selection flag on all rows which are matched.

By default the number of selected rows is unlimited. This can be explicitly requested by setting the first optional parameter to a negative value. Any positive value stops the execution of the command after the specified number of matched rows have been processed. The second optional parameter is the index of the first row where processing starts. The default is zero, meaning that the table scan is started at the top.

The expression argument is an **SQL** expression which checks cell values. The syntax of this expression is that of the *where* clause of an **SQL** statement. The syntax is compatible to the **SQL** dialect used in the **MYSQL** database. Almost all **SQL** functions that database provides in the 4.1 release can be used here, including aggregate functions and function columns which are dynamically computed. However, only columns in the local table can be used. Cross-referencing to other tables is not supported.

Example:

```
table sqlselect $thandle "E_WEIGHT>avg(E_WEIGHT) and E_NROTBONDS<5"
```

where `E_WEIGHT` and `E_NROTBONDS` are either the names of existing table columns, or the table has retained active row references to ensembles. In case a reference is used, the data is obtained indirectly by reading existing property data from the linked ensemble, or even by computing it on the fly. This type of ensemble- or reaction-linked computation is only possible for column names which can be identified as properties. Existing table columns which are used in the expression can be property columns, function columns, or simple data columns. The use of existing columns has precedence over references, even if the cell data is, for example, **`NULL`** and not identical to what would be obtained from the structure object.

Ensemble or reaction references in rows are conveniently introduced by using **`table addens`** or **`table addreaction`** commands to fill the rows. They are silently lost if the ensembles or reactions which provided the data are deleted from memory. It is possible to save tables in the native **CACTVS** format with all reference objects, so that they could be restored from a file, but this is not the default. Normally, tables lose object references when written to file and re-read.

The result is a list of the row indices of all selected rows after the operation.

This command is very powerful, but can be overkill in many circumstances. For a simpler, and in many cases faster, alternative refer to the **`table select`** command.

table sqlshow

```
table sqlshow tablehandle propertylist ?filterset? ?parameterlist?
t.sqlshow(property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **`table get`** command. The differences between **`table get`** and **`table sqlshow`** are that the latter does not attempt computation of property data, but raises an error if the data is not present and valid, and that the **SQL** command variant formats the data as **SQL** values rather than for **TCL** or **PYTHON** script processing.

table string

```
table string tablehandle ?format? ?parameter value?...
table string tablehandle ?format? ?dictionary?
t.string(?format?,?parameter,value?,...)
t.string(?format?,?dict?)
```

Encode the contents of the table as a blob image of an file of the specified format. Except that the output goes to a blob instead of a file, this command is the same as **`table write`**.

The result value is a byte vector, not a string, so this command can be used with binary table formats.

table subcommands

```
table subcommands
dir(Table)
```

Lists all subcommands of the **`table`** command. Note that this command does not require a table handle.

table transfer

```
table transfer tablehandle propertylist ?targethandle? ?targetpropertylist?
t.transfer(properties=,?target=?,?targetproperties=?)
```

Copy property data from one table to another table or other major object, without going through an intermediate scripting language object representation, or dissociate property data from the table. If a property in the argument property list is not already valid on the source table, an attempt is made to compute it.

If a target object is specified, the return value is the handle or reference of the target table. The source table and the target object cannot be the same object.

If a target property list is given, the data from the source is stored as content of a different property on the target. For this, the data types of the properties must be compatible, and the object class of the target property that of the target object. No attempt is made to convert data of mismatched types. In case of multiple properties, the source property list and the target property list are stepped through in parallel. If there is no target property list, or it is shorter than the source list, unmatched entries are stored as original property values, and this implies that the object class of the source and target objects are the same.

If no target object is specified, or it is spelled as an empty string or **Python** **`None`**, the visible effect of the command is the same as a simple **`table get`**, i.e. the result is the property data value or value list. The property data is then deleted from the source object. In case the data type of the deleted property was a major object (i.e. an ensemble, reaction, table, dataset or network), it is only unlinked from the source object, but not destroyed. This means that the object handles or references returned by the command can henceforth the used as independent objects. They can be deleted by a normal object deletion command, and are no longer managed by the source object.

table transfercolumn

```
table transfercolum tablehandle srcolumn dstcolumn ?target?
t.transfercolumn(srccolumn=,dstcolumn=,?target=?)
```

Transfer column data to another data column, or auxiliary cell data field. The data transfer operation attempts casting of the source value to the datatype of the destination - either the column datatype, or a string for the auxiliary cell values. If a cast fails, the cell data is a **`NULL`** value without an error message. The original data remains unchanged. The source and destination column may be identical, and this can be useful if the target is not the cell value. When casting to a string for the auxiliary cell fields, the string is formatted according to the format set on the table, column, row and cell level.

The transfer target can either be *data* (the column data, which is the default), *comment* (cell comment) or *linktext* (the text output instead of the actual underlying link destination when writing table files which support links, such as **HTML**, **MS Word** or **PDF**.

The return value is the table handle or reference.

table trim

```
table trim tablehandle ?positions?
t.trim(?positions=?)
```

Remove leading or trailing table rows or columns where all cells have **`NULL`** values. If no position argument is specified, both rows and columns are processed. This is equivalent to the positions argument *all*. Alternatively, location values *columns* (only columns are processed), *rows* (only rows are processed), or a list of *left* (leading columns), *right* (trailing columns), *top* (leading rows) and *bottom* (trailing rows) are recognized.

This command only processed leading or trailing rows or columns. Empty rows or columns which are embedded in the middle of the table are not removed.

This command is useful for quickly cleaning up tables with spurious extra empty rows or columns, as they are often found in Excel spread sheets and similar sources.

The command returns the original table handle or reference.

table unlock

```
table unlock tablehandle propertylist/table/all
t.unlock(property=)
```

Unlock property data for the table, meaning that they are again under the control of the standard data consistency manager.

The property data to unlock can be selected by providing a list of the following identifiers:

- Property names
  Valid property instances on the table are unlocked. Non-existent data is silently ignored. It is not possible to unlock individual property fields.
- *all*
  All valid table properties are unlocked.
- *table*
  This is an object class identifier. All property data which is controlled by the table major object and attached to the specified object class is unlocked. Since tables currently do not have minor objects, this command does the same as the *all* variant.

Property data locks are obtained by the **`table lock`** command.

The return value is the original table handle or reference.

table unpack

```
table unpack datastring ?compressionlib?
Table.Unpack(data=,?compressionlib=?)
```

Unpack a base64-encoded serialized object string which was created by a **`table pack`** command. The return value of this function is the handle or reference of the newly created table object, which is an exact duplicate of the packed original table.

Packed tables may also be unpacked by the **`table create`** command.

The default compression library is *zlib*. For more options, see **`table pack`**.

table valid

```
table valid tablehandle propertylist
t.valid(property/propertysequence)
```

Returns a list of boolean values indicating whether values for the named properties are currently set for the table. No attempt at computation is made.

Example:

```
table valid $thandle T_COMMENT
```

reports whether the table has a comment property attached or not.

`table has` is an alias to this command.

table varexport

```
table varexport tablehandle ?varname? ?reset?
t.varexpor(?variable=?,?reset=?)
```

Export the table contents into a global **Tcl** array or **Python** dictionary variable in the current thread-local main interpreter. If the boolean reset flag is given as *true*, any previous variable contents are erased. If the variable name is not specified, the variable name set via the *variable* table attribute (see **`table get`** command) is used. If the variable name is unset or an empty string, the command does nothing. The *variable* table attribute is updated if a variable name is supplied.

The rows and columns of the table are translated into a comma-separated numerical index in the array variable, i.e. the cell at the first row and column in the table has the array index or dictionary key “0,0”. The addressing is scheme is row-first.

Within the **Tcl** interface, this command establishes a variable trace link between the variable and the table. Any data updates in either the table or the **Tcl** variable is automatically reflected in the other object.

The command returns the original table handle or reference.

table varimport

```
table varimport tablehandle ?varname?
t.varimport(?variable=?)
```

Import cell data from a global **Tcl** array variable or **Python** dictionary in the current thread-local main interpreter. The array variable is expected to use row and column addressing with row-first numerical, comma-separated indices (see **`table varexport`**), and to contain element data which is compatible with the column data types of the table.

If the variable name is not specified, it is taken from the *variable* table attribute (see **`table get`** command). If the variable name is not set, or an empty string, the command does nothing. In any case, the *variable* table attribute is not changed, and, for **Tcl**, no automatic update link between the **Tcl** variable and the table is established. Existing table cells for which no array variable element exists remain unchanged. Array variable elements which do not correspond to existing table cells are ignored.

Example:

```
set tvar(0,0) "data_for_row0_col0"
set tvar(0,1) "data_for_row0_col1"
table varimport $thandle tvar
```

If the *tvar* variable did not exist before, and the table has at least one row and two columns, and these cells can accept string data, the first two table cells in row zero are updated.

The use of this command with variables which are linked to the table object (via **`table varexport`** or **`table set`** commands) is not required, since data changed a linked variable is automatically propagated.

The command returns the original table handle or reference.

table verify

```
table verify tablehandle property
t.verify(property)
```

Verify the values of the specified property on the table. The property data must be valid, and a table property. If the data can be found, it is checked against all constraints defined for the property, and, if such a function has been defined, is tested with the value verification function of the property.

If all tests are passed, the return value is boolean 1, 0 if the data could be found but fails the tests, and an error condition otherwise.

table wait

```
table wait tablehandle conditionflags
t.wait(conditionflags)
```

Suspend execution of the current thread until one or more conditions on the table have been met. While waiting, the mutex lock on the table is released, so that other threads can manipulate it. While a wait operation is in progress, the table cannot be deleted.

The flags may be a list of one or more of the following flags, plus *none* or an empty string (and **`None`** for **Python**) if no flags should be set:

- *eod*
  The end-of-data condition has been reached (see the *eod*, *targeteod* and *checkeod* table attributes)
- *hasrows*
  The table has one or more rows of data.
- *norows*
  The table contains no (more) rows
- *script*
  A script embedded in an **RPC** input stream was received. This flag is reset when an **RPC** transfer for the table begins, and set when either the script was received, or a status flag that no script will me sent in the transmission. In the latter case, the *importscript* attribute of the table is set to an empty string. Scripts are always transmitted after the table column definition and other structure information has been sent, but usually before all cell data has been transmitted.

table write

```
table write tablehandle ?filename? ?format? ?parameter value?..
table write tablehandle ?filename? ?format? ?dictionary?
t.write(?filename?,?format?,?parameter,value?,...)
t.write(?filename?,?format?,?dict?)
```

Save the contents of the table to a file. If no filename is specified, or an empty string, `#auto` or `None` for **Python**, the same filename as that the table was originally read from, or configured via a `table set` command, is used. If this name is not set, a name with a standard suffix located in the temporary directory is generated, but this requires that the *format* argument is set. Besides a normal file name, standard output and standard error (*stdout*, *stderr*) as well as, except on Windows, an open **Tcl** or **Python** file or socket handle may be specified. If the file name begins with a vertical bar, a pipe channel is opened. If the file name is specified, it is remembered as future default.

If no explicit table file format is specified, an attempt is made to infer the format from the file name suffix. If necessary, table I/O modules are loaded automatically. If none of these measures are able to automatically choose an format, the original table format is used if it is defined. If still no table file format can be identified, an error is reported. Otherwise, the selected file format is remembered as future default.

The rest of the optional arguments are either keyword/value pairs (an even number of additional command arguments) or a single dictionary argument with the same possible set of keyword/value pairs. The following keywords are recognized:

- *append*
  A boolean value of 1 as attribute value instructs the table I/O module to append the data, not to rewrite the file or database table. This is currently supported only for a few table formats, such as database table I/O modules which do not delete and recreate the destination table in this mode.
- *compression*
  Select the compression method of the table file format if it supports different types of compression. For example, **ORC** table files can use compression methods *none*, *zlib* or *snappy*. The allowed compression schemes depend on the format, and most formats do not use compression, or utilize a fixed method. In these cases, the value is ignored.
- *colblocksize*
  The column block size. specified as an integer of 1 (the default, no column blocking) or more. Not many table formats support the combination of columns into blocks. This is the same attribute as can be set or retrieved for the table as a table-level attribute, but if used in this context, it is not permanently changed. After the command completes, the old value is restored.
- *colnames*
  A boolean flag indicating whether the column names shall be included in the output or not. This flag is only honored when the file format allows a choice whether to include these or not.
- *columns*
  The argument is a column range list. Only the selected columns are output. By default, all columns are written. The alias *cols* is also recognized.
- *compact*
  A boolean attribute. If set, the output is formatted in a compact style, if the format supports distinct compact/compressed and expanded/beautified/human-readable representations. For most formats, this flag is ignored.

- *corsdomain*
  If specified, and the *httpheader* attribute is also set to a value larger than zero, the output **HTTP** header contains a **CORS** domain header line (*Access-Control-Allow-Origin:*). Useful values for this parameter are either * (for free access), a host name, or a domain name. If this parameter is not specified, **HTTP** headers do not contain information, which usually is equivalent to allowing access for **AJAX** queries only from the same server as the requesting page.
- *debug*
  A boolean flag indicating that, if set, debugging information should be printed while writing the file. This is primarily a developer option.
- *downloadfilename*
  If a **HTTP** header is configured (see *httpheader* attribute), a download target file name is included in the **HTTP** header if the attribute is not set to an empty string. By default, the value of this attribute is copied from the persistent *downloadfilename* table attribute. If it is neither set there, nor as transient write attribute, no save file name is transmitted.
- *embedfileformat*
  The format of embedded objects in the table. This is equivalent to setting the same attribute with a **`table set`** command. It is explained in the paragraph about the **`table get`** command.
- *extsjsstatus*
  This flag only applies to the *extjs* table file format. If set, the content is wrapped in an **XML** dictionary with keys *success* set to true, and the table data under key *data*. This format is expected by the **ExtJS** Web toolkit in response to **AJAX** submissions.
- *firstrow*
  The first row (in zero-based index notation) to output. By default, output starts at row zero.
- *flatten*
  A boolean flag indicating whether the data of columns of complex data types such as vectors and compound properties should be temporarily expanded into columns of simpler data types. Please refer to the **`table flatten`** command for more details. In any case, this variant of the flattening operation is transient and the table is restored to the same column set after the write as it had when the command was started. The default value for this option is dependent on the selected output format. If output is in a format which does not support vector data in cells, flattening is performed by default. For output in vector-capable formats, column flattening is disabled by default.
- *formatbits*
  A collection of additional formatting bits to be applied to the output besides the format-dependent standard set. This is not a commonly used attribute, and not every format bit is compatible with every output format. Example:

```
table write $th1 kinome_export.xlsx xlsx {formatbits unit|rescale}
```

In this example, the Excel cells in the output do not only contain the cell values, but additionally (for those columns which are associated with a property, which then also needs to possess a specified unit) the unit name as a string, and an attempt is made to adjust the Excel cell value to lie between 1 and 1000 with automatic adjustment of the unit within a known unit series, for example by switching between the unit names *pMol*, *nMol*, *uMol*, *mMol* and *Mol*.

- *frame*
  A boolean value indicating whether certain graphical representations of structures or reactions should be wrapped in a frame or not. Examples are embedded **OLE** objects in **XLSX** and **RTF** output. By default, a frame is drawn around the structure or reaction rendering in these formats, ensuring, for example, uniform size and centering of the images in the output file. This option has only an effect if the rendering is produced implicitly at output time. If cell data has been explicitly set to, for example, an `E_GIF` or `E_EMF_IMAGE`, the presence or absence of a frame in the drawing is not checked and must have been configured by suitable property computation parameters when the image was computed. It is also possible to retain an invisible frame by setting its color to the background color, see *framecolor* table object attribute.

- *httpheader*
  Configure whether the table data content output should be prefixed by a **HTTP** header. This can be useful when scripting **CGI** or **FCGI** applications. The value can be 0 (no **HTTP** header prefix, the default), 1 (standard **HTTP** header prefix, without status code) or 2 (**HTTP** header including 200 status code). String values *none*, *default* and *status* are also recognized. Not all table I/O modules support this feature.

- *maxrows*
  An integer indicating the maximum number of rows to write. A negative value (the default) indicates that there is no set limit.

- *mode*
  The output mode for the table, interpreted like the standard text file output modes. The value can be either *w* or *a*, and is ignored for most table file formats. The only exception are those formats which can store more than one table in the file. For these files, the current table may either be added or replaced in mode *a* (depending on the sheet name), or a new file is written with the current table as only data in mode *w*. The default value of this option is *w*, and it is not supported for direct string output (**`table string`** command), where the implied value is always *w*.

- *monochrome*
  If this boolean value is set, the output of structure and reaction drawings is configured to be monochrome, i.e. use only black and white rendering colors. This applies only to renderings which are created at the moment the file is written (e.g. **OLE** objects or generic image class columns), but not if, for example, a **GIF** or **EMF** image has been set as explicit cell data - these are not re-computed. Cell property data formatting colors are not modified by this flag. The values of the *framecolor* and *markcolor* table attributes are ignored when the flag is set.

- *oleautoupdate*
  This boolean value only applies to output formats which include Microsoft OLE objects. Such files can be written in two styles: With the pre-rendered display (a WMF image) and the silent OLE data backing it, so that it is only updated when the OLE application supporting the embedded data format is started by a double click, or auto-updating, which means that the OLE application is automatically started when the document is opened and

the image is re-rendered immediately. Generally, the pre-rendered images are of inferior quality than the native Windows re-rendered images, but re-rendering a lot of OLE objects in a document can take significant time. This option, which is by default off, lets you control the update style.

- *overspill*
  A boolean flag which can be set to prevent distributing tables onto multiple pages if the rendering of the table is larger than the configured paper size. This option has an effect only for output formats which use the concept of a paper as drawing area, such as **RTF** or **CDXML** and generally provide means to manually reformat the table in the native editing program (e.g. **MS WORD** for **RTF** or **CHEMDRAW** for **CDXML**, but not **PDF**).

- *pagebreaks*
  A boolean attribute which influences the placement of page breaks in some output formats, for example **CDXML** tables. If set, a new page is started at the end of a column block in rotated mode. By default, the current page is filled until no additional row or column can be placed.

- *paperorientation*
  The orientation of the paper for print output, specified as *portrait* or *landscape*. The default is *portrait*. This is not the same as the *rotate* attribute, which controls the layout of the table data, not the dimensions of the output medium. Only output formats which use paper-based formatting (currently this is jut **PDF**) use this attribute. The parameter may also be abbreviated to *orientation*.

- *papersize*
  The size of the paper, specified as a standard name such as *A4* or *letter*, to be used for formatting. This attribute is currently only used for **PDF** output and has no effect for other formats. This parameter may be abbreviated to simply *paper*. If this parameter is not set, the default paper size, which is configurable via the *::cactvs(paper_size)* control variable, is used.

- *recodebitvectors*
  A boolean value indicating whether a bit vector should be recoded on output to an integer vector with zero bit padding for bits filling up a full integer value instead of writing a true bitvector. This is only supported on a few output formats, most notably SQL database formats.

- *rotate*
  A boolean flag which will, if set, virtually rotate the table by 90 degrees counterclockwise for output, thus swapping rows and columns. This is supported only for a few formats, such as **HTML** and **EXCEL**. The table object in memory does not change by this operation. The attribute controls the layout of the table output, not the dimensions of the output medium (see *paperorientation* parameter).

- *rownames*
  A boolean flag indicating whether the row names shall be included in the output or not. This flag is only honored when the file format allows a choice whether to include these or not.

- *rows*
  The argument is a row range list. Only the selected rows are output. By default, all rows are written.

- *sheetname*
  The name of the sheet the current table is stored in, if the selected output file format supports this concept. The default sheet name is dependent on the output format.
- *structures*
  A boolean flag indicating whether structures (ensembles and reactions) associated with the table rows should be included in the file. This option has an effect only for a few formats where this is supported, for example the native binary **CACTVS** table format. Only the primary row object is written - if cell data has been set from other structure objects, their association is not remembered and they are not included in the output.

`table save` is a command alias.

table yield

```
table yield tablehandle ?waitusecs?
t.yield(?waitusecs=?)
```

This command is primarily useful in multi-threaded application. It temporarily sets the table undeletable, releases all mutex locks, instructs the thread scheduler to yield the current thread, and them either immediately or after the specified number of microseconds re-acquires the mutex locks and restores the original object deletability.

The return value of the command is the original table handle or reference.

A typical application is within a **`table loop`** or **`table poploop`** command where the processing of a row by the loop command is faster than the delivery of new table rows by a separate thread. Using this command can lead to a better distribution of table access time between the loop command and the data provision thread and an overall speed-up of the computation.

The *vertex* Command

The vertex command is used to access information about vertices in generic network objects (see **`network`** command). In many respects the behavior of vertex objects in networks is comparable to that of atoms in ensembles, and the commands for handling vertices are similarly structured.

Pseudo vertex labels *first*, *last* and *random* are special values, which select the first vertex in the vertex list, the last, or a random vertex.

The command *node* is an alias for *vertex*, allowing the use of a more standard nomenclature, but without the benefit of a matching prefix on the names of vertex properties. The prefix for vertex properties is `V_`.

The following vertex commands are supported:

vertex append

```
vertex append nhandle label ?property value?...
v.append({?property:value,?...})
v.append(?property,value,?...)
```

Standard data manipulation command for appending property data. It is explained in more detail in the section about setting property data.

vertex children

```
vertex children nhandle label ?filterset? ?filtermode? ?sphere? ?allowduplicates?
v.children(?filters=?,?mode=?,?sphere=?,?allowduplicates=?)
```

This is a variant of the **`vertex neighbors`** command, with the additionally applied constraint that all matching vertices must have a value of property `V_LEVEL` which is exactly higher by one than that of the originating vertex.

The command parameters are explained in the paragraph on command **`vertex neighbors`**.

The **`vertex parents`** command can be used to find parents instead of children.

vertex connections

```
vertex connections vhandle label ?filterset? ?filtermode?
v.connections(?filters=?,?mode=?)
```

Standard cross-referencing command to obtain the labels or references of the connections the vertex is participating in. This is explained in more detail in the section about object cross-references.

Example:

```
vertex connections $nh $v
```

vertex create

```
vertex create nhandle ?property value?...
Vertex(nref,?property,value?...)
Vertex(nref,dict)
Vertex.Create(nref,?property,value?...)
Vertex.Create(nref,dict)
```

Create a new vertex in the existing network object. The new vertex is created without any connections or initial data except for its automatically assigned label.

An initial set of vertex properties can be set by the optional property/value arguments.

The commands returns the label or reference of the new vertex.

vertex defined

```
vertex defined nhandle label property
v.defined(property)
```

This command checks whether a property is defined for the vertex. This is explained in more detail in the section about property validity checking. Note that this is *not* a check for the presence of property data! The **`network valid`** command is used for this purpose.

vertex delete

```
vertex delete nhandle ?label?...
vertex delete nhandle all
v.delete()
Vertex.Delete(nref,?label?,...)
Vertex.Delete(vref,...)
Vertex.Delete(nref,"all")
```

Delete specific or all vertices and all connections a deleted vertex is participating in.

The command returns the number of deleted vertices.

vertex dget

```
vertex dget nhandle label propertylist ?filterset? ?parameterdict?
v.dget(property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **`vertex get`** command. The difference between **`vertex get`** and **`vertex dget`** is that the latter does not attempt computation of property data, but rather initializes the property values to the default and return that default if the data is not yet available. For data already present, **`vertex get`** and **`vertex dget`** are equivalent.

vertex exists

```
vertex exists nhandle label ?filterlist?
v.exists(?filters=?)
Vertex.Exists(nref=,label=,?filters=?)
```

Check whether this vertex exists. Optionally, a filter list can be supplied to check for the presence of specific features. The command returns 0 if the vertex does not exist, or fails the filter, and 1 in case of successful testing.

Example:

```
vertex exists $nh 99
```

vertex filter

```
vertex filter nhandle label filterlist
v.filter(filters)
```

Check whether a vertex passes a filter list. The return value is boolean 1 for success and 0 for failure.

Example:

```
filter create rootnode property V_LEVEL value 0 operator =
vertex filter $nh $v rootnode
```

vertex get

```
vertex get nhandle label propertylist ?filterset? ?parameterdict?
v.get(property=,?filters=?,?parameters=?)
v[property]
v.property
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

Examples:

```
vertex get $nhandle 1 {V_LABEL V_IDENT}
```

yields the label and ID data of vertex 1 as a list. If the information is not yet available, an attempt is made to compute it. If the computation fails, an error results.

```
vertex get $nhandle 1 C_ONTOLOGY_LINK
```

gets the ontology link types of all connections the vertex participates in.

For the use of the optional property parameter list argument, refer to the documentation of the **`ens get`** command.

Variants of the **`vertex get`** command are **`vertex new, vertex dget, vertex nget, vertex show, vertex sqldget, vertex sqlget, vertex sqlnew`** and **`vertex sqlshow`**.

Further examples:

```
vertex get $nhandle 1 V_ONTOLOGY_TERM(structurehash)
```

vertex index

```
vertex index nhandle label
v.index()
```

Get the index of the vertex. The index is the position in the vertex list of the network. The first position is index 0.

Example:

```
vertex index $nhandle 99
```

vertex jget

```
vertex jget nhandle label propertylist ?filterset? ?parameterdict?
v.jget(property=,?filters=?,?parameters=?)
```

This is a variant of **`vertex get`** which returns the result data as a **JSON** formatted string instead of **TCL** or **PYTHON** interpreter objects.

vertex jnew

```
vertex jnew nhandle label propertylist ?filterset? ?parameterdict?
v.jnew(property=,?filters=?,?parameters=?)
```

This is a variant of **`vertex new`** which returns the result data as a **JSON** formatted string instead of **TCL** or **PYTHON** interpreter objects.

vertex jshow

```
vertex jshow nhandle label propertylist ?filterset? ?parameterdict?
v.jshow(property=,?filters=?,?parameters=?)
```

This is a variant of **`vertex show`** which returns the result data as a **JSON** formatted string instead of **TCL** or **PYTHON** interpreter objects.

vertex neighbors

```
vertex neighbors nhandle label ?filterset? ?filtermode? ?sphere? ?allowduplicates?
v.neighbors(?filters=?,?mode=?,?sphere=?,?allowduplicates=?)
```

This command (which can also be invoked as subcommand **`neighbours`**) is a cross-referencing command with some extra options and, in some filter modes, slightly different behavior than the standard object cross-reference subcommands.

In the simplest case, it returns the labels or references of the immediate neighbor vertices. A neighbor vertex is a vertex which is linked via a connection to the originating vertex. In case the filter list contains connection filters, the connection leading to the originating vertex must pass the check, not just any connection of the neighbor vertex.

Example:

```
filter create onto_isa property C_ONTOLOGY_LINK value is_a operator =
vertex neighbors $nhandle 1 onto_isa
```

returns a list of neighbor vertex labels which are linked via a connection which encodes an ontological “is a” definition. Neighbor vertices which participate in such a link with another vertex, but not the originating vertex, are not returned.

This command supports two special *filtermode* values in addition to the standard set (*exists, count, exclude, include*). The *incoming* mode only sees links where the originating or previous vertex is the last vertex in the vertex list of any passed connection. The *outgoing* mode only sees links where the originating or previous vertex is the first vertex in the vertex list of any passed connection.

Example:

```
vertex neighbors $nh $vlabel {} outgoing
```

only returns the links where the originating vertex is the first vertex in the connection linking the originating vertex to its neighbor. In applications where the connections are set up in a directional fashion, or example in networks representing an ontology with “is a” relationships, this mode allows the traversal of the network in the direction of increasingly generic terms.

By default vertices in the immediate neighborhood are examined, but this change be changed by the *sphere* parameter. The immediate neighbors are in sphere 1 (the default for this parameter), the next group of vertices is in sphere 2, and so on. If the sphere is not 1, the special filtering of connections is no longer active and the normal object substitution mechanism for cross referencing is used. When going beyond the first sphere, it is also possible that a vertex may be reached by multiple paths of the selected length. By default, these vertices are returned only once, but with the last optional parameter this behavior may be changed.

A positive sphere value only selects vertices in that sphere. A negative sphere parameter value returns a list of all neighbors up to and including the sphere identified by the absolute parameter value.

Example:

```
vertex neighbors $nh $vlabel {} outgoing -5
```

Above example reports the labels of all vertices of to a distance of 5 steps from the starting point which can be reached via *outgoing* directional connections.

vertex network

```
v.network()
```

Python-only method to get the network reference from a vertex reference.

vertex new

```
vertex new nhandle label propertylist ?filterset? ?parameterdict?
v.new(property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **`vertex get`** command. The difference between **`vertex get`** and **`vertex new`** is that the latter forces the re-computation of the property data, regardless whether it is present and valid, or not.

vertex nget

```
vertex nget nhandle label propertylist ?filterset? ?parameterdict?
v.nget(property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **`vertex get`** command. The difference between **`vertex get`** and **`vertex nget`** is that the latter always returns numeric data, even if symbolic names for the values are available.

vertex parents

```
vertex parents nhandle label ?filterset? ?filtermode? ?sphere? ?allowduplicates?
v.parents(?filters=?,?mode=?,?sphere=?,?allowduplicates=?)
```

This is a variant of the **`vertex neighbors`** command, with the additional automatically applied constraint that all matching vertices must have a value of property `V_LEVEL` which is exactly one less than that of the originating vertex.

While it would often be expected that there is only a single parent to a vertex, this is not a condition enforced in a generic network, and potentially multiple parents can be found and reported by this command. In any case, the command alias **`vertex parent`** in singular is also recognized.

The command parameters are explained in the paragraph on command **`vertex neighbors.`**

The **`vertex children`** command can be used to find children instead of parents.

vertex paths

```
vertex paths nhandle label targetlabel ?parameterdict?
v.paths(target=,?parameters=?)
```

This command finds paths between a pair of vertices, traversing connections.

The return value of the command is a nested list, even it only a single path is found. Every sublist contains all the labels (if this is not overridden in the parameter dictionary) of the vertices in one path, including those of the start and end vertices. Every connection is used only once in any path, and no path crossings through a vertex are allowed. Every vertex, with the possible exception of path end points, appear only once in any single path. Paths from a vertex via some connections back to itself are allowed. The vertex must be a member of a cyclic connection for such paths to exist.

If the destination vertex is specified as an empty string, all possible paths emerging from the source vertex and not violating any other specified constraints are returned. This includes shorter sub-paths which are contained in a longer paths - these are reported as separate result items.

The optional parameter dictionary can be used to further customize the path walking and result reporting. The following dictionary keywords are recognized:

- *filters*
  A filter set specification to limit the set of vertices and/or connections which can be a part of the path. The default filter set is empty and all vertices and connections can be used in the path.
- *flags*
  A collection of keywords which further modify path traversal and result reporting. Currently, the following flag words are recognized:

  none
  Equivalent to not setting any flag

  ascending
  Property `V_LEVEL` must increase on each path node, but not necessarily in an uniform manner.

  descending
  Property `V_LEVEL` must decrease on each path node, but not necessarily in an uniform manner.

  constant
  Property `V_LEVEL` must be constant on path nodes

  different
  Property `V_LEVEL` must be different from node to node. The same value may be encountered at multiple nodes, as long as these are not direct neighbors in the path.

  concatenate
  Return path as concatenated string, not as list elements

outgoing
Path connections must be traversed from first to second node in connection definition. This option is useful only if the connections have been set up in a directional fashion.

incoming
Path connections must be traversed from second to first node in connection definition. This option is useful only if the connections have been set up in a directional fashion.

rootfinder
Path connections must first be traversed in incoming or outgoing direction, and then in reverse direction to the target node. The point of inversion can be anywhere in the path, but there can only be one.

minlength
Only report the found paths of minimum length, and those that are the minimum length or less after the *lengthfuzz* parameter is subtracted, if one was specified. This flag bit is not identical to the parameter of the same name in the general parameter dictionary.

maxlength
Only report the found paths of maximum length, and those that are the maximum length or more after the *lengthfuzz* parameter is added, if one was specified. This flag bit is not identical to the parameter of the same name in the general parameter dictionary.

endpoints
Only report the terminal nodes of found paths, as simple list elements, not the complete path. In addition, reporting of terminal nodes which are contained in the full paths of longer matches are suppressed. The encounter check uses vertex identities, not the properties which are reported in the result. For example, if the result set consists of vertex labels 1-2, 1-2-3, and 1-2-4-5, the result is a list with elements 3 and 5.

- *length*
  A specific length all reported paths must have. This is equivalent to setting the *maxlength* and *minlength* parameters to the same value.
- *lengthfuzz*
  A fuzz value to allow paths slightly longer than the minimum or slightly shorter than the maximum length to be still reported, if the *minlength* or *maxlength* options are set in the *flags* parameter. The default fuzz is zero.
- *maxlength*
  The maximum length of reported paths. The default maximum path length is 30. The path length is defined as the number of nodes in the path.
- *maxpathcount*
  The maximum number of paths returned. If there are more paths, traversal is stopped early. The default value is -1, indicating an unlimited number of reported paths. The order in which paths are found depends on the internal representation of the network and is not guaranteed to be canonic.
- *minlength*
  The minimum path length of reported paths. The default minimum path length is 2, i.e. a path cannot consist only of the start node.

- *property*
  The name of the property whose values are reported as path elements. The default is `V_LABEL`, the standard vertex identifier. The use of a property name with a field instead of a simple property name is also supported.

Example:

```
vertex paths network0 $v1 $v2 \
   [dict create property V_ONTOLOGY_TERM(id) flags outgoing]
```

This example returns all paths from network vertex *v1* to *v2* in outgoing direction. The reported path data is the value of field *id* of property `V_ONTOLOGY_TERM` on every vertex in the paths.

vertex ref

```
Vertex.Ref(nref,identifier)
```

Python only method to get a vertex reference. See **`vertex vertex`** command.

vertex set

```
vertex set nhandle label ?property value?...
v.set(?property,value?,...)
v.set({property:value,...})
v.property = value
v[property] = value
```

Standard data manipulation command. It is explained in more detail in the section about setting property data.

Example:

```
vertex set $nhandle 1 V_IDENT “V1294”
```

vertex show

```
vertex show nhandle label propertylist ?filterset? ?parameterdict?
v.show(property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **`vertex get`** command. The difference between **`vertex get`** and **`vertex show`** is that the latter does not attempt computation of property data, but raises an error if the data is not present and valid. For data already present, **`vertex get`** and **`vertex show`** are equivalent

vertex sqldget

```
vertex sqldget nhandle label propertylist ?filterset? ?parameterdict?
v.sqldget(property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **`vertex get`** command. The differences between **`vertex get`** and **`vertex sqldget`** are that the latter does not attempt computation of property data, but initializes the property value to the default and returns that default, if the data is not present and valid; and that the **SQL** command variant formats the data as **SQL** values rather than for **Tcl** or **Python** script processing.

vertex sqlget

```
vertex sqlget nhandle label propertylist ?filterset? ?parameterdict?
v.sqlget(property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **`vertex get`** command. The difference between **`vertex get`** and **`vertex sqlget`** is that the **SQL** command variant formats the data as **SQL** values rather than for **TCL** or **PYTHON** script processing.

vertex sqlnew

```
vertex sqlnew nhandle label propertylist ?filterset? ?parameterdict?
v.sqlnew(property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **`vertex get`** command. The differences between **`vertex get`** and **`vertex sqlnew`** are that the latter forces re-computation of the property data, and that the **SQL** command variant formats the data as **SQL** values rather than for **TCL** or **PYTHON** script processing.

vertex sqlshow

```
vertex sqlshow nhandle label propertylist ?filterset? ?parameterdict?
v.sqlshow(property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **`vertex get`** command. The differences between **`vertex get`** and **`vertex sqlshow`** are that the latter does not attempt computation of property data, but raises an error if the data is not present and valid, and that the **SQL** command variant formats the data as **SQL** values rather than for **TCL** or **PYTHON** script processing.

vertex subcommands

```
vertex subcommands
dir(Vertex)
```

Lists all subcommands of the **`vertex`** command. Note that this command does not require a network handle or a vertex label.

vertex vertex

```
vertex vertex nhandle label
Vertex.Ref(nref,identifier)
```

Return the vertex label stored in property `V_LABEL` (**TCL**) or a minor object reference (**PYTHON**). This is useful in case the label used in the command is not a straightforward numerical label or reference but some other vertex identification format.

Substructure Match Commands

Substructure and reaction substructure matching is a complex functionality. While a limited number of object commands are supplied (**`ens match`**, **`mol match`**, etc.), comprehensive match functionality is accessible via special commands. The match command implements various structure matching commands.

The match command

The **`match`** command matches substructures, maximum common substructures, and reactions. Its syntax scheme is

```
match ss ?-option value?... ss_spec st_spec
    ?atommatchvar? ?bondmatchvar? ?molmatchvar?
```

```
match mcss -option value?... ss_spec st_spec
    ?atommatchvar? ?bondmatchvar? ?molmatchvar?
```

```
match rss -option value?... rxnss_spec rxn_spec ?reagent_atommatchvar?
    ?reagent_bondmatchvar? ?reagent_molmatchvar? ?product_atommatchvar?
    ?product_bondmatchvar? ?product_molmatchvar?
```

```
match(class='ss'/'mcss'/'rss',substructure=,structure=,?align=?,
    ?allowmissingstereo=?,?anchor=?,?atomenvproperty=?,?atomhighlight=?,
    ?atomlistcomparison=?,?atommapproperty=?,?atommatchcount=?,
    ?bondenvproperty=?,?bondhighlight=?,?bondmapproperty=?,?burnmode=?,
    ?chain=?,?charge=?,?clearatomhighlight=?,?clearbondhighlight=?,
    ?cmpflags=?,?cmpflags2=?,?command=?,?creategroup=?,
    ?daylightaromaticity=?,?excludeenvironment?,?excludeflags_ss=?,
    ?excludeflags_st=?,?excludeh=?,?excludelabels_ss=?,?excludelabels_st?=,
    ?excludelabels_st_root=?,?excludestructures=?,?exclude_ss_h=?,
    ?exclude_st_h=?,?fixedframework=?,?forceringmatch=?,?formula=?,?fuzz=?,
    ?generalize=?,?heavyatoms=?,?hinstances_ss=?,?implicit_is_singlearo=?,
    ?includeflags_ss=?,?includeflags_st=?,?includelabels_ss=?,
    ?includelabels_st=?,?isotope=?,?kekule=?,?limit=?,?mapanchor=?,
    ?maxopenlinks=?,?mode=?,?multihighlight=?,?noaliphaticonaro=?,?noarofg=?,
    ?nochainonaro=?,?nochainonring=?,?nodoubleonaro=?,?noheterofg=?,
    ?nomultibondfg=?,?nosingleonaro=?,?nosuperatommultilink=?,
    ?nosuperatomonh?,?omitrecursion=?,?openhcount=?,?overlap=?,?pionaro=?,
    ?queryatomexpansion=??remembercomplexmatches=?,?restrictsubstitution=?,
    ?rotateterminals=?,?stereo=?,?strictexclusion=?,?strictsmarts=?,
    ?superatomexpansion=?,?superatommapproperty=?,?tautomers=?,
    ?terminal=?,?timeout=?,?transferstereo=?,?useatomtree=?,?useatomtype?,
    ?usebondorder=?,?usebondtree=?,?varbondglobal=?,?varbondlocal=?,?wedge=?,
    ?atommatchvariable=?,?bondmatchvariable=?,?molmatchvariable=?,
    ?product_atommatchvariable=?,?product_bondmatchvariable=?,
    ?product_molmatchvariable=?)
```

The first word defines the match class and also the type of the expected structure or reaction arguments. Match class *ss* performs a substructure match operation, class *mcss* a simple maximum common substructure match operation, and class *rss* a reaction substructure match. In the latter case, the substructure and structure arguments are expected to be reaction substructures (for example, **SMIRKS**) and reaction handles or specifications (for example, reaction **SMILES**) instead of simple structures.

Structure and substructure may both be independently specified in several different formats:

- An ensemble handle or reference

  This is probably the most common format. It is allowed to match a structure or a part of it onto itself. Example:

  ```
  match ss $ss_handle $st_handle
  ```

  For reaction substructure matching, the argument should be a reaction handle or reference.

- A list of an ensemble handle and a molecule label, or a molecule reference in **PYTHON**.

  This short-cut selects one molecule from the substructure or structure ensemble for matching. All other molecules in the ensemble are ignored. Example:

  ```
  match ss $ss_handle [list $st_handle 1]
  ```

  This mode is not applicable to reaction substructure matching.

- A **SMILES** or **SMARTS** string.

  The argument is interpreted as a **SMARTS** expression, with full query feature support and without implicit hydrogen addition for the substructure argument and as standard **SMILES** for the structure argument. The temporarily decoded structure is automatically destroyed after the command has completed. However, for the sake of performance, decoded substructure **SMARTS** structures are cached, so that it is only a small performance hit if the same substructures are repeatedly passed to this command. The exact size of the **SMILES/SMARTS** cache varies, but is usually around 500 substructures. Example:

  ```
  match ss c1ccccc1 $st_handle
  ```

  For reaction substructure matching, the argument is expected to be reaction **SMILES** or **SMIRKS** string instead.

- Another standard string-based structure representation

  This includes packed ensemble serialized ensemble (or reaction, for *rss* matching) object strings, and hex-encoded **SMILES**. These specifications are essentially managed like **SMILES/SMARTS** strings, but the decoded structures are not cached.

The return value of the command is the number of successful matches. For simple match modes, which return only a *match/nomatch* result, this is 0 or 1, but modes which can produce multiple matches may return higher counts.

The final three optional parameters are names of variables which receive atom, bond and molecule mapping information. If these parameters are not supplied, or a variable name is spelled as an empty string (or **None** for **PYTHON**), no variable is created or modified. If a variable is specified, but no match is found, the map variables are set to empty strings. For reaction matching, there are separate variable sets for the reagent and product side. In Python reaction matching the standard three variables apply to the reagent side, while the product side is addressed with an explicit argument name prefix i.e. , **atomapvariable=** vs. **product_atommapvariable=**. The variable named in parameter **atommapvariable** is used both for normal substructure and reagent-side reaction match information.

For match modes which can only return a single match, these map variables are simple nested lists. Each list element contains a substructure and a structure object label or reference, in this order. The number of elements in the result list corresponds to the number of substructure objects. Example:

```
match ss CN CCN amap bmap mmap
```

The variable **amap** is set to "**{1 2} {2 3}**", which is the first match of the C-N substructure fragment on the ethylamine structure. The numbers are atom labels - in case of **SMILES** strings, atom labels are assigned in the order the atoms appear in the string. The **bmap** variable is set to "**{1 2}**", since only a single bond is involved. The first bond of the substructure matches the second bond of the structure. Finally, the **mmap** variable is set to "**{1 1}**", because both substructure and structure contain only a single molecule, which was assigned the default label 1. The bond and mol map results are still nested lists, even if they appear in this simple example as plain lists

There is no guarantee that the lowest possible labels are use for a simple match - the match algorithm uses internal optimizations for choosing good start atoms for matches. Matches should not be expected to start with an atom with the lowest label. Match result variables are filled in the order of the objects in the internal object lists, which also is not necessarily an ascending label sequence.

These nested result lists can easily be transformed to an **Tcl** array with a statement like

```
array set array_amap [join $amap]
```

The array variable **`array_amap`** now contains elements which are named with the substructure labels, and have values which correspond to the structure labels. The **`unzip`** command is also useful to isolate substructure or atom label sets.

In case a match mode is invoked which can return more than one match, the map variables are constructed with an additional nesting level. They are a list, where each element describes one match. Each of these elements for a specific match is formatted as in above description of simple match results. Note that the actual number of reported matches does not influence the scheme - if there is a theoretical possibility that more than one match can be found, the maximum nesting level is 3, not 2, even if only a single match is finally found.

Example:

```
set nmatches [match -mode distinct CC CCC amap]
```

Here, the match count is 2 (the *distinct* mode reports matches which differ by at least one structure atom from any previous match - the *all* mode reports 4 matches, which include reversals of the CC fragment), and the **`amap`** variable is set to **`"{{1 1} {2 2}} {{1 2} {2 3}}"`**. The first match is substructure atom 1 on structure atom 1, and 2 on 2, the second match maps substructure atom 1 on structure atom 2, and substructure atom 2 on structure atom 3.

The **`match ss`** command has a large number of options, which can be used to fine-tune the matching process. In the **Tcl** version, any number of options, in any order, may be inserted before the substructure specification. In the **Python** version, they are specified as named arguments. Every argument after the structure must use the named argument style.

This is the list of options:

-align

```
-align none/rotate/redraw/xaxis/yaxis/diagonal/combined
```

If a match was successful, change the layout of the *structure* by modifying the `A_XY` atomic 2D coordinates property.

Mode *none*, which is the default, does not perform any 2D coordinate changes.

In modes *xaxis*, *yaxis* and *diagonal*, the coordinates of the matched *structure* atoms are extracted and the largest principal component/eigenvector of these computed. The structure is then rotated in such a way that this eigenvector is aligned to the x-axis, y-axis, or diagonal (lower left to upper right). No coordinates of the *substructure* atoms are used.

In *rotate* mode, the *structure* is rotated in steps of 15 degrees, with and without a flip. The orientation which is in best alignment with the coordinates of the matched *substructure* atoms is retained.

In *redraw* mode, the *structure* is completely redrawn, using the coordinates of the matching *substructure* atoms as starting point. The rest of the *structure* is drawn around it. The matched structure atoms possess the same coordinates as the matching substructure atoms.

There are some limitations in this mode, which are automatically enforced by setting the corresponding match control flags. First, it is not possible to match partial ringsystems. A substructure ring atom must match the same class of ring system, i.e. a substructure 6-membered ring fragment only matches a structure benzene or cyclohexane ring, but not naphthalene, adamantane, etc. This limitation is deeply rooted in the 2D layout generator, which treats ring systems different from the acyclic connections. Acyclic substructure atoms can only match acyclic structure atoms, with the exception that a *terminal* acyclic substructure atom may still match a structure ring atom.

The final mode *besteffort* combines the *redraw* and *rotate* modes - if a match in mode *redraw* fails, the match attempt is automatically repeated in mode *rotate*, which has relaxed match conditions with respect to ring system checks.

-allowmissingstereo

```
-allowmissingstereo none/atoms/bonds/both
```

If not set to *none*, the default, stereogenic atom or bond centers on the structure side may be matched by corresponding centers on the substructure with defined stereochemistry if they do not possess a non-zero stereo descriptor in `A_LABEL_STEREO` or `B_LABEL_STEREO`. If there is a structure-side stereo descriptor on the matched center, the normal stereo match process applies (i.e. absolute or relative stereo matching). This is a global option which applies to the complete substructure pattern. There are also atom- and bond-specific bits in `A_QUERY` and `B_QUERY` to control this feature on a local level in the pattern.

-anchor

```
-anchor nested_anchor_object_list
```

This option defines restrictions on which substructure and substructure atoms or bonds must match. The argument is a nested list where each outer list element is a list of two or three elements.

The first two of the inner list elements are either an object label, an empty string, or the word *any.* The latter two options are equivalent. The first sublist item identifies a substructure object, the second a structure object. If no third sublist argument is supplied, the object labels are interpreted as atoms. With the third optional argument, an explicit minor chemistry object classification such as *atom* or *bond* can be set, though currently only anchor atoms and bonds are supported.

If *any* or an empty identifier is used as sublist argument, it is read as a wildcard. If two object labels are paired, the two objects must map onto each other in all reported matches. If these objects are incompatible, no matches are found. If a wildcard is used, it means that the partner object must be in the match, but without the need to match any specific counter object. If fuzzy matching is used, this can make sense even on the structure side. The use of a pair of wildcards is not illegal, but has no filtering effect.

Example:

```
match ss -anchor {{1 2} {any 3}} $sshandle $ehandle
```

This sample line only returns a match where (in addition to meeting the query conditions of the substructure ensemble object) substructure atom 1 maps onto structure atom 2, and structure atom 3 is part of the match without requesting it to be mapped to any specific substructure atom.

```
match ss -anchor {{1 2 bond}} $sshandle $ehandle
```

This command only reports matches where substructure bond 1 is mapped to structure bond 2.

-atomenvproperty

```
-atomenvproperty 0/1
```

If set, fill property `A_SSMATCH_ENVIRONMENT` on match. This is an experimental feature.

-atomhighlight

`-atomhighlight` *`none/structure/substructure/both`*

If this flag is set, all matched atoms in the structure (modes structure or *both*, or numeric encodings 1 or 3) or substructure (modes *substructure* or *both*, or the equivalent numeric encodings 2 or 3) have the *highlight* flag set in property `A_FLAGS`. In case multiple matches are generated, the result depends on the **`-multihighlight`** option setting. By default, only the first match is highlighted, but highlighting the union of all found matches is also possible. This option does not reset existing atom highlight flags - see the **`-clearatomhighlight`** option for this functionality. By default this function is disabled (equivalent to mode *none* or 0).

-atomlistcomparison

```
-atomlistcomparison intersection/identity
```

Specify how the matching of atom list should be handled if they are encountered on both the structure and substructure sides. By default any common element in these lists is sufficient for a match (*intersection* mode), but in *identity* mode they must contain the same set of elements.

-atommapproperty

`-atommapproperty` *`none/structure/substructure/both`*

If this flag is set, for each match a new instance of property `A_SSMATCH` is attached to the structure (in modes *structure* or *both*, or numeric encodings 1 or 3) ensemble, or a new instance of property `A_STMATCH` to the substructure (in modes *substructure* or *both*, or the equivalent numeric encodings 2 or 3) ensemble - the first match is recorded in `A_SSMATCH` or `A_STMATCH`, the second in `A_SSMATCH/2` or `A_STMATCH/2`, and so on. If instances of this property are already set on the structure or substructure ensembles, the new instances start with the highest existing instance number plus one. Structure or substructure atoms which are not used in a match have their respective `A_SSMATCH` or `A_STMATCH` data set to 0. Matched structure atoms are marked with the atom label of the matching substructure atom, and matched substructure atoms with the atom label of the matching structure atom. By default, this flag is not active (equivalent to mode *none* or 0).

-atommatchcount

`-atommatchcount none/structure/substructure/both`

Specify whether the number of time an atom was matched in successful matches should be recorded in properties `A_SSMATCH_COUNT` (for the structure side) or `A_STMATCH_COUNT` (for the substructure side).

-bondenvproperty

`-bondenvproperty 0/1`

If set, fill property `AB_SSMATCH_ENVIRONMENT` on match. This is an experimental feature.

-bondhighlight

`-bondhighlight none/structure/substructure/both`

If this flag is set, all matched bonds in the structure (modes *structure* or *both*, or numeric encodings 1 or 3) or substructure (modes *substructure* or *both*, or the equivalent numeric encodings 2 or 3) have the *highlight* flag set in property `B_FLAGS`. In case multiple matches are generated, the result depends on the **`-multihighlight`** option setting. By default, only the first match is highlighted, but highlighting the union of all found matches is also possible. This option does not reset existing bond highlight flags - see the **`-clearbondhighlight`** option for this functionality. By default this function is disabled (equivalent to mode *none* or 0).

-bondmapproperty

`-bondmapproperty` *`none/structure/substructure/both`*

If this flag is set, for each match a new instance of property `B_SSMATCH` is attached to the structure (in modes *structure* or *both*, or numeric encodings 1 or 3) ensemble, or a new instance of property `B_STMATCH` to the substructure (in modes *substructure* or *both*, or the equivalent numeric encodings 2 or 3) ensemble - the first match is recorded in `B_SSMATCH` or `B_STMATCH`, the second in `B_SSMATCH/2` or `B_STMATCH/2`, and so on. If instances of this property are already set on the structure or substructure ensembles, the new instances start with the highest existing instance number plus one. Structure or substructure bonds which are not used in a match have their respective `B_SSMATCH` or `B_STMATCH` data set to 0. Matched structure bonds are marked with the bond label of the matching substructure bond, and matched substructure bonds with the bond label of the matching structure bond. By default, this flag is not active (equivalent to mode *none* or 0).

-burnmode

`-burnmode` *`modeflags`*

This flag has no effect in normal stand-alone substructure matches. Its settings are relevant in case prior substructure matches to exclude certain areas of the structure from matching are spliced into the processing before the actual substructure match. This mode defines how the exclusion match effects remaining area for the final match. Thius is a bit set and multiple flags can be combined. Possible values are *atoms* (mark all matched atoms are excluded), *bonds* (mark matched bonds), *carbon* (mark matched carbon atoms, but not the rest of the matched structure part), *terminals* (mark matched terminal atoms), *ringsystems* (mark all complete ringsystems where at least one atom was matched) and *aroringsystems* (mark the connected aromatic part of ringsystems where at least one ring atom was matched).

-chain

```
-chain 0/1
```

If set, this flag allows additional matches after the first match only if these matches are chained to a previous match, i.e. they do not overlap with any previous match, but a *normal* or *complex* bond exists between at least one structure atom of the new match and a structure atom of a previous match. In more complex cases, the results of this command variant can depend on the atom order. For example in case of a structure which contains a left part *A* and a right part *AA* linked by some construct, matching with substructure fragment *A* returns a single hit if the left part is matched first, but two fragment matches if the right part is matched first. However, within a single chain of building blocks in the structure it does not matter where the first match occurs - the chain fragment is recursively appended in all directions and ultimately cover all linked blocks. The chain does not need to be linear - rings or star topologies can be matched, too. Obviously, this option has no effect in match mode *first*, because specific results are only generated when more than a single match is sought.

-charge

```
-charge 0/1
```

This flag determines whether atomic formal charges on the substructure and substructure atoms are used for determining the possibility of an atom match. By default, formal charges are ignored. This option only affects the standard match attributes. Atom query expressions which explicitly refer to property `A_FORMAL_CHARGE` always use their comparison result to determine matches.

-clearatomhighlight

```
-clearatomhighlight 0/1
```

If this flag is set, all *highlight* bits in property `A_FLAGS` are reset on the structure (and possibly the substructure) ensemble before the first match is processed. By default, this flag is not set and any existing `A_FLAGS` *highlight* bit pattern remains unchanged. Because the reset is performed in the routine where the highlight bits are set, this option is effective only in combination with the **`-atomhighlight`** option. The decision whether to reset the flags on the structure or substructure side, or both sides, follows the setting of the **`-atomhighlight`** mode.

-clearbondhighlight

```
-clearbondhighlight 0/1
```

If this flag is set, all *highlight* bits in property `B_FLAGS` are reset on the structure (and possibly the substructure) ensemble before the first match is processed. By default, this flag is not set and any existing `B_FLAGS` *highlight* bit pattern remains unchanged. Because the reset is performed in the routine where the highlight bits are set, this option is effective only in combination with the **`-bondhighlight`** option. The decision whether to reset the flags on the structure or substructure side, or both sides, follows the setting of the **`-bondhighlight`** mode.

-cmpflags

```
-cmpflags flags
```

This option provides a direct access to the full set of flags which modify the substructure match process. The more common flags can be set or unset with specific options of this command for convenience. The default flag set is *bondorder|useatomtree|usebondtree.*

The flag set can either override the default flags (if specified as simple attribute list), added to them (if prefixed with a ’+’), removed, (if prefixed with a ’-’), or toggled (if prefixed with a ’^’).

These are generally useful flags recognized:

- *none*
  No flags
- *arotautomer*
  See *tautomer* flag.
- *atomlistcontained*
  When this flag is set and matching a structure side atom list onto a substructure side atom list, the match is only successful if all elements of the structure list are also listed in the substructure list. The substructure list may contain additional elements. By default, an single common element between the lists is sufficient for a positive match. This flag cannot be used together with *atomlistidentity.*
- *atomlistidentity*
  When this flag is set and matching a structure side atom list onto a substructure side atom list, the match is only successful if the lists contain the same set of allowed elements, though they do not need to be listed in the same order. By default, an single common element between the elements is sufficient for a positive match. This flag cannot be used together with *atomlistcontainer.*
- *bondorder*
  Match bond order of non-aromatic bonds. This flag should usually be set.
- *bondreaction*
  Match reaction query bond attributes against structure property `B_REACTION_CENTER`, if present. By default, bond query attributes are not checked for match in substructure matching, but they are tested in reaction matching.
- *chained*
  If multiple matches are found, matches after the first must be aligned in such a way that they are adjacent to an atom matched by a previous match. This flag is used to extract polymer patterns. The use of this flag is explained in more detail in the paragraph on the **`-chain`** option.
- *charge*
  Match the formal charge of atoms. Query charges specified in `A_QUERY` field or part of a atom query expression are always matched, but formal charges on atoms in `A_FORMAL_CHARGE` are not by default.

- *daylight*
  Use daylight aromaticity for matching, regardless of the currently configured global aromaticity system. This is equivalent to using the *-daylightaro* option and explained in more detail in its paragraph.
- *excludesmartsenvironment*
  Standard recursive **SMARTS** matches have no knowledge which atoms and bonds were already matched in upper recursion levels, and all neighbor atoms are again available for re-matching in the new recursion level. If this flag is set, atoms which are already matched by upper recursion level are no longer eligible for re-matching and are effectively removed from the structure visible to the new level.
- *fixedframework*
  If set, the matched structure part cannot be bonded to unmatched non-hydrogen atoms. See also *terminal* for more flexibility.
- *generalizeheteroatoms*
  Any substructure atom which is not carbon or hydrogen matches any structure atom which is not carbon or hydrogen.
- *implicitissinglearo*
  If set, all bonds which were encoded with an implicit bond order, for example when decoding **SMARTS** strings, and where this bond status is registered in `B_QUERY(`*flags*`)`, are matched as if they were explicitly specified as a *single_or_aromatic* query bond. This matches the default behavior of Daylight-conforming **SMARTS** matching.
- *isotope*
  Substructure atoms with an isotope label only match structure atoms with of the same isotope. By default, isotope labels are ignored in matching.
- *kekuleeven*
  The Kekule bond orders of even-membered aromatic rings must match that of the substructure. By default, the formal bond order of aromatic systems is not checked. See also the **`-kekule`** option for additional information.
- *kekuleodd*
  The Kekule bond oder of odd-membered aromatic rings must match that of the substructure. By default, the formal bond order of aromatic systems is not checked. See also the **`-kekule`** option for additional information.
- *matchallheavyatoms*
  If set, the substructure must cover all non-hydrogen atoms of the structure ensemble for a successful match.
- *matchatomringcount*
  The ring count (in the **ESSSR**) of the substructure atoms must be the same as that of matched structure atoms. This prevents matching of substructures that are embedded in larger ring systems.

- *matchbondringcount*
  The ring count (in the **ESSSR**) of the substructure bonds must be the same as that of matched structure bonds. This prevents matching of substructures that are embedded in larger ring systems.
- *matchfullens*
  Matching substructure must match all atoms in the ensemble, including all hydrogen atoms.
- *matchfullmolecule*
  A matching substructure must match all atoms of a fragment, including all hydrogen atoms. There may be unmatched fragments present in the structure ensemble.
- *matchfullringsystem*
  If set, the substructure must match any ring systems of the structure completely or not at all, but not partially. Every ring system is tested independently.
- *no3dcoordinatecomputation*
  When performing a 3D match, no attempt should be made to compute 3D atomic coordinates of structure ensembles if they do not yet possess them. Instead, the 3D match immediately fails.
- *noaliphaticonaro*
  If set, aliphatic substructure atoms cannot match aromatic structure atoms, even if the other match flags indicate that aromaticity should not be checked.
- *noalkyllink*
  This is an additional criterion for the *terminal* match flag. If it is set, the single allowed non-hydrogen bond leading from the the matched structure part to the unmatched structure part cannot lead to an unsubstituted carbon with only carbon and hydrogen neighbors.
- *noarobondfg*
  If set, carbon structure atoms which participate in an aromatic bond are not considered functional groups. This has an effect on matching special pseudo-atoms, such as *insulator* or *terminator* fragments.
- *noatomstereook*
  If set, query atoms with specified atom stereochemistry, and general stereochemistry match options active, can match a structure atom without stereochemistry, but not with different stereochemistry. For individual atoms, this can be configured in `A_QUERY`.
- *nobondstereook*
  If set, query bonds with specified atom stereochemistry, and general stereochemistry match options active, can match a structure bond without stereochemistry, but not with different stereochemistry. For individual bonds, this can be configured in `B_QUERY`.
- *nochainonaro*
  Substructure bonds that are not ring bonds cannot match aromatic structure bonds.
- *nochainbondonring*
  Substructure bonds that are not ring bonds cannot match structure ring bonds.
- *nochainatomonring*
  Substructure atoms that are not ring atoms cannot match structure ring atoms.

- *nochargepaircollapse*
  Control the match feature that bonds which connect two atoms with opposite formal charges +1 and -1 also match an increased bond order, e.g. a substructure ionic nitro group also matches a pentavalent structure group, and vice versa. This feature is enabled by default, setting this flag switches it off.
- *nodoubleonaro*
  If set, non-aromatic double query bonds do not match aromatic structure bonds. By default they do.
- *noheterofg*
  f set, carbon structure atoms which participate in a bond connecting to a hetero atom are not considered functional groups. This has an effect on matching special pseudo-atoms, such as *insulator* or *terminator* fragments.
- *nomultibondfg*
  If set, carbon structure atoms which participate in a multiple, non-aromatic bonds are not considered functional groups. This has an effect on matching special pseudo-atoms, such as insulator or terminator fragments.
- *nosingleonaro*
  If set, single query bonds do not match aromatic structure bonds. By default they do.
- *nosmallerring*
  A matched structure atom cannot be a member of a ring of size 8 or smaller if this ring is smaller than the smallest ring of the matching substructure atom.
- *nosuperatomonh*
  Superatoms (like Beilstein query atoms) in the substructure cannot match a single hydrogen in the structure. Expanding superatoms matching multiple structure atoms, such as *alkyl*, still match hydrogens attached to inner atoms, such as the hydrogen atoms in a methyl group.
- *nullsubstructureismismatch*
  If this flag is set, a substructure without any atoms does not match any structure. By default, it reports a successful match.
- *openhcount*
  If set, all hydrogen ligand count query attributes are interpreted as lower limits, not as exactly required values.
- *openlinkrequiresheavyatom*
  An open link pseudo-atom can on
- ly match a heavy atom, not hydrogen.
- *piaroatoms*
  If set, structure atoms which possess π electrons and are ring atoms are considered aromatic, even if they are not a member of a full aromatic system.

- *relativestereo*
  If set, stereo query atoms outside explicit stereo groups can match either enantiomer, but only in a synchronized fashion, e.g. an R/S or S/R configuration, but not S/S or R/R. This flags implies setting the *stereo* flag to switch on general stereochemistry matching.
- *relaxedmatchatomringcount*
  A variant of *matchatomringcount*. A substructure atom can only match if it has the same ESSSR ring count in structure and substructure, or if the substructure atom is not a ring atom and simultaneously a terminal atom. This rather exotic option is used internally for 2D drawing template alignment.
- *relaxedmatchbondringcount*
  A variant of *matchbondringcount*. A substructure bond can only match if it has the same ESSSR ring count in structure and substructure, or the substructure bond is not a ring bond, and at least one of the atoms of the bond is a terminal atom. This rather exotic option is used internally for 2D drawing template alignment.
- *remembermatches*
  This is an optimization flag. If set, once a match or mismatch between a substructure and substructure atom or bond has been established. it is remembered and not re-checked during atom-by-atom matching. This flag is incompatible with some advanced match options (such as context-sensitive custom callback functions, or **SMARTS** environment manipulations), but safe for standard **SMARTS** or **MDL**-style structure matching. In that case, time savings can be significant especially in match modes returning more than one match.
- *restricthydrogenmatches*
  If set, an explicit hydrogen atom on a substructure fragment can only match hydrogen atoms on the structure which have the same ligand sphere hydrogen index. This avoids the permutation of many hydrogen mappings in match modes which return more than one result. For example, a substructure with two explicit hydrogen atoms bonded to a central carbon atom matches a structure methyl group in six different mapping without this flag, but only one with it. In that mapping, the first substructure H ligand matches the first structure H ligand, and the second substructure H ligand the second structure H ligand. The matching of hydrogen substructure atoms onto different structure atoms is not affected. When using this flag, the true equivalence of the matched structure atoms is not tested - so if isotope labeling of structure H atoms or their 3D coordinates are of significance for the match, it should not be used.
- *restrictanyatommatches*
  If set, *any* query atoms occurring simultaneously on the substructure and structure sides cannot match each other. They each can only match atoms in the other fragment which are not of the *any* query type.
- *restrictsubstitution*
  If set, unmatched non-hydrogen portions of the structure can only be linked to substructure *any* atoms, or open valences. If open valences are used, the number of open valences linked to a substructure atom limits the number of unmatched heavy atom continuation atoms on the structure. Example: A substructure *phenyl-any* would match benzene (on a hydrogen substituent), toluene or other mono-substituted benzenes, but not higher-substituted compounds. A substructure *open*-C-*open* would not match tertiary or quaternary structure carbons.

- *singleattachmentsuperatomsonly*
  If set, expanding superatoms can only match structure parts which are connected by a single bond to the matched core.
- *singleonany*
  If set, only multiple, non-aromatic bond s in the query need to match the bond order of the structure. Single bonds of the substructure match any structure bond order.
- *stereo*
  Match stereochemistry. In the absence of a set flag for relative stereochemistry (*relativestereo*) matching this selects matching of absolute stereochemistry. Explicit absolute/relative stereochemistry match instructions, for example via **MDL** stereo groups, are always matched as specified. This flag only applies to specified atomic stereo centers outside query stereo groups.
- *strictexclusion*
  See **`-strictexclusion`** option.
- *tautomer*
  Match tautomeric structure forms. For this, the substructure must contain mobile hydrogens which can be linked to a tautomeric systems in the substructure. If this is the case, and the structure also contains tautomeric parts, the bond orders in the tauto systems need only to match as a sum of bond orders. The hydrogen is not fixed and may match a structure part which is distant from its original attachment point. This version of tautomer matching does not allow tautomer systems to traverse and implicitly dissolve or create aromatic systems. The *arotautomer* flag is a more aggressive variant which allows this.
- *terminal*
  A successful match has at most as as specified in the *maxopenlinks* parameter (default is 1) substructure atoms where its matched structure atom has bonds to unmatched structure parts that are not simple hydrogen atoms, and there is at maximum one bond to an unmatched non-H atom on the structure continuation atoms.
- *terminalbondorder*
  If set, bond order matching is only performed for structure bonds which are terminal, i.e. where one of the atoms of the bonds only participates in a single bond.
- *unsetaromaticisaliphatic*
  Any substructure atom which is neither in a complete auto-identified aromatic system, nor has an *aromatic* `A_QUERY` attribute implicitly bears an *aliphatic* query attribute.
- *useatomtree*
  heck the full atom attribute query expression tree if present for an atom. If not set, only the flat **MDL**-style single-level attribute set without logic is used. This flag should be set for normal queries.
- *usebondtree*
  Check the full bond attribute query expression tree if present for a bond. If not set, only the flat **MDL**-style single-level attribute set without logic is used. This flag should be set for normal queries.

- *varbondglobal*
  If this flag is set, and a parameter for the maximum allowed deviation of fractional bond orders in structure property `B_ORDER_ESTIMATE` is set, the average absolute deviation of the found structure bond orders in the matched part versus the requested value in non-zero `B_QUERY(`*varbo*`)` substructure values is checked and the match fails if it exceeds the threshold. The **`-varbondglobal`** command option sets this flag as a side effect. f that option is not used, the default maximum deviation parameter is 0.0, i.e. it is very strict

- *varbondlocal*
  If this flag is set, and a parameter for the maximum allowed deviation of fractional bond orders in structure property `B_ORDER_ESTIMATE` is set, a bond match fails if the absolute difference of the fractional bond order in structure property `B_ORDER_ESTIMATE` versus the requested query bond order in a non-zero `B_QUERY(`*varbo*`)` value exceeds the threshold. The **`-varbondlocal`** command option sets this flag as a side effect. If that option is not used, the default maximum deviation parameter is 0.0, i.e. it is very strict

- *wedge*
  Match the presence and style of wedge bonds as graphical attributes. This is not the same as matching stereochemistry, since a common stereochemical configuration can have multiple valid wedge representations!

-cmpflags2

Another set of comparison flags. The default is an empty flag set. Recognized flags are:

- *equivalentonly*
  If the match mode allows the reporting of multiple matches, and there is more than one pattern match, the match command returns success only if all matches are topologically equivalent. If there is any other match, the complete result set is scrapped and the command returns zero matches, but no error.

- *nonequivalentonly*
  If the match mode allows the reporting of multiple matches, and there is more than one pattern match, the match command returns success only if all matches are topologically different. If there is any other match, the complete result set is scrapped and the command returns zero matches, but no error.

- *requirecarbonmatch*
  The substructure must match at least one simple carbon atom.

- *requireheteromatch*
  The substructure must match at least one simple hetero atom (no carbon, no hydrogen).

- *screening*
  Internal use

-command

```
-command tcl_command/python_function
```

Define a Tcl or Python callback function which is called when a new match is found and all property-based constraints have been checked. This function is called with four parameters. The first two parameters are the handles or references of the substructure and structure ensembles.

The third parameter is a nested list of label/reference pairs (substructure atom label/structure atom label or references) for all substructure atoms which are currently matched to a structure atom. The fourth parameter is a nested list of label/reference pairs (substructure bond label/structure bond label or references) for all substructure bonds which are matched to a structure bond. The format of these arguments is the same as that of the match variables of the **match** command for single-match modes. Within the callback functions, the match can be further evaluated in ways not possible by the standard match options.

If the function returns 0, any non-numeric value, or throws an error, the post-processing of completed matches, such as atom or bond highlighting, is not executed and the match discarded.

While the callback routine is free to perform any additional match analysis, it must neither delete the structure or substructure, nor change its connectivity (remove or add atoms and bonds), nor discard or invalidate any property data used in the matching process. The computation or setting of additional property data on the substructure or structure ensembles is allowed.

By default, or in case an empty string is passed as callback procedure name, no callback is executed.

Example:

```
proc my_match_check {ens_ss ens_st amap bmap} {
   puts $amap
   return 1
}
match ss -command my_match_check CC CC
```

This example outputs “{1 1} {2 2}”, which is the atom mapping of the match found.

-creategroup

```
-creategroup 0/1
```

If this flag is set, every match creates a new group minor object on the structure ensemble. The atoms in the group are all those structure atoms which were matched by the substructure. The group name (property `G_NAME`) is set to the name of the substructure (property `E_NAME`). By default, no groups are generated as side effects of a match.

-daylightaromaticity

```
-daylightaromaticity 0/1
```

If the flag is set, the use of **DAYLIGHT** aromaticity in the matching is enforced both on the structure and substructure side regardless of the global aromaticity system setting. For the substructure, this applies to implicitly defined aromaticity, for example the presence of a complete aromatic ring with all defined bond orders and elements, not explicit query attributes. The option can be shortened to -*daylightaro*.

-excludeenvironment

```
-excludeenvironment 0/1
```

If this flag is set and a recursive **SMARTS** expression is processed, all parts of the structure which are already matched are excluded from the recursive match check. By default, a new recursion level does not have any knowledge about previous matches and may match all atoms in the structure.

Example:

```
match ss -excludeenvironment 0 {C[$(OC)]} CO
match ss -excludeenvironment 1 {C[$(OC)]} CO
```

The first example does match, because the carbon of the recursive fragment may match on the same structure carbon as the first carbon atom in the substructure. In the second case, the structure carbon is marked as already matched, and there is no place to map the recursive fragment carbon, so no match is found.

-excludeflags_ss

```
-excludeflags_ss flag_value
```

This option allows the exclusion of substructure atoms from the match procedure which have at least one of potentially several bits set in the `A_FLAGS` property. The decoded flag values are used as a bit mask, and all structure atoms which have one or more bits of the mask set are hidden from further processing.

Example:

```
match ss -excludeflags_ss [list starred boxed] $ss_handle $st_handle
```

This example ignores all substructure atoms which have been marked with the *starred* or *boxed* attribute.

All substructure atom exclusion options can be combined, but not repeated, and are cumulative.

-excludeflags_st

```
-excludeflags_st flag_value
```

This option allows the exclusion of structure atoms from the match procedure which have at least one of potentially several bits set in the `A_FLAGS` property. The decoded flag values are used as a bit mask, and all structure atoms which have one or more bits of the mask set are hidden from further processing.

Example:

```
match ss -excludeflags_st [list starred boxed] $ss_handle $st_handle
```

This example ignores all structure atoms which have been marked with the *starred* or *boxed* attributes.

All structure atom exclusion options can be combined, but not repeated, and are cumulative.

-excludelabels_ss

```
-exclude_ss label_list
-excludelabels_ss label_list/aref_sequence
```

This option allows the exclusion of a set of substructure atoms from the match process. All atoms which are listed here are completely ignored by the match algorithm. By default, or when an empty list is passed, all substructure atoms of the ensemble or molecule (if the handle/molecule label specification was used) are used for matching.

```
Example:
match ss -exclude_ss [ens atoms $sshandle hydrogen] $sshandle $sthandle
```

This example does not use any hydrogens on the substructure for matching. This is more efficient and stripping and possibly re-attaching the hydrogen atoms from the substructure.

All substructure atom exclusion options can be combined, but not repeated, and are cumulative.

-excludelabels_st

```
-exclude_st label_list
-excludelabels_st label_list/aref_sequence
```

This option allows the exclusion of a set of structure atoms from the match process. All atoms which are listed here are completely ignored by the match algorithm. By default, or when an empty list is passed, all structure atoms of the ensemble or molecule (if the handle/molecule label specification was used) are available for matching.

All structure atom exclusion options can be combined, but not repeated, and are cumulative.

-excludelabels_st_root

```
-exclude_st_root label_list
-excludelabels_st_root label_list/aref_sequence
```

This set of structure atoms to be excluded is similar to the one specified with **`-exclude_st`**. The difference is that this exclusion only applies to the first level of matching. In deeper match levels, for example recursive **SMARTS** expressions, these atoms are no longer blocked.

All structure atom exclusion options can be combined, but not repeated, and are cumulative.

-excludestructures

```
-excludestructures ens_mol_list
```

Specify of set of exclusion fragments. These structure fragments are exhaustively matched as substructures on the structure, and all structure atoms and bonds they match are excluded from the actual match procedure invoked by this command. The exclusion fragment substructure match is always performed with the default mode settings - options like **`-bondorder`** or **`-charge`** are only applied to the final match. The exclusion fragments may be specified in the same styles as the main substructure and structure, i.e. as an ensemble handle, a list of an ensemble handle and a molecule label, or as a **`SMILES/SMARTS`** string.

Example:

```
match ss {[OH]} CC(=O)O
match ss -excludestructures {C(=O)[OH]} {[OH]} CC(=O)O
```

The first example matches the hydroxyl group of the structure, which is acetic acid. In order to prevent of match of hydroxyl groups which are part of carboxylic acid groups, carboxylic acid groups can be ignored on the structure with a statement like in the second example. Of course, this example could be easily made more generic, such as hiding all groups which have the hydroxyl group attached to any non-carbon, or carbon with any other hetero atom neighbor, as in

```
match ss -excludestructures {[!C,C&x{2-}][OH]} {[OH]} $sthandle
```

All structure atom or fragment exclusion options can be combined, but not repeated, and are cumulative.

-exclude_ss_h

```
-exclude_ss_h 0/1
```

If this flag is set, all substructure hydrogen atoms are ignored in the match process. By default, all atoms in the substructure are used.

All substructure atom exclusion options can be combined, but not repeated, and are cumulative.

-exclude_st_h

```
-exclude_st_h 0/1
```

If this flag is set, all structure hydrogen atoms are ignored in the match process. By default, all atoms in the structure are used.

All structure atom exclusion options can be combined, but not repeated, and are cumulative.

-fixedframework

```
-fixedframework 0/1
```

If this flag is set, all *carbons* in the structure are prevented from possessing any unmatched hetero atom or carbon neighbors. Matched structure hetero atoms may be bonded to unmatched hetero atoms or carbon atoms. By default, the flag is not set. The acceptability of extra unmatched hydrogen, carbon, or hetero atom neighbors may be additionally controlled on the atomic level by setting the appropriate flags in property `A_QUERY`(*flags*) on the substructure.

Example:

```
match ss -fixedframework 1 CC CCO
match ss -fixedframework 1 CCO CCOC
```

The first example does not match, because in all possible match orientations there is one matched carbon with bonded to an unmatched hetero atom (the oxygen atom). The second example does match - the matched hetero atom may possess bonds to unmatched non-hydrogen atoms - the methyl group in this case.

This match option is useful for locating starting materials for synthesis in vendor catalogs.

-forceringmatch

```
-forceringmatch no/strict/relaxed/ringsystem
```

This option controls the matching of the substructure into structure ring systems. If the option is not specified, or set to *no* (or 0), the matching is only controlled by explicitly set atom and query attributes, such as the number of ring bonds, or membership of rings of specific size.

The option value *strict* allows the matching of substructure atoms or bonds which are members of rings only onto structure parts in ring systems of the same class, i.e. the same set of rings of a given size and arrangement, but without consideration of atoms, bond orders, aromaticity, etc. With this option, a phenyl substructure fragment no longer matches a naphthalene structure, and acyclic substructure atoms or bonds can only match acyclic structure parts. All other query attributes, such as bond order, element type, aromaticity, etc. are applied in addition to this constraint.

The *relaxed* mode has basically the same constraints, but with one small exception: A terminal substructure atom (an atom which has only a single bond, and thus cannot be a ring member) may match onto structure atoms in ring systems, if the normal query attributes allow this.The *relaxed* mode is automatically enforced if the **-align** option with value *redraw* is specified.

The *ringsystem* mode requires that any structure ringsystem is either completely matched, or not part of the match at all.

-formula

```
-formula formulaexpression
```

Require that the set of structure atoms matched by the substructure also matches a formula expression (see **molfile scan**). The matched atom set is tested after full expansion. This means that multiple structure atoms which are matched by a single query atom (for example, an **[ALK]** alkyl superatom group) all count. However, structure atoms which are matched multiple times (for example, by overlapping fragments with a suitable overlap mode) are only tallied once. The default other element mode for the formula expression is *exact*, but that can be changed by the operator prefix (see again **molfile scan**).

Example:

```
match ss -formula C1-2N1-2 {[C,N][C,N][C,N]} $ehstructure
```

This example will only allow substructure matches where the substructure matches 1-2 carbon and 1-2 nitrogen atoms but where the relative orientation of the matched atoms is arbitrary. It disallows a C_3 or N_3 match, which would be allowed by the **SMARTS** string.

-fuzz

```
-fuzz n
```

If this option is used with a value *n* larger than zero, fuzzy substructure matching is activated. In this mode, it is no longer required that all substructure atoms are mapped to structure atoms. Up to *n* atoms may fail. Within the `A_QUERY` property, fields are provided which allow a more detailed specification whether a substructure atom may be in the fail set, and how much fuzz is allowed in its immediate neighborhood. The **-anchor** option is also useful to force the use of some critical substructure atoms in the found matches.

This match variant is computationally significantly more expensive than the standard match procedure, and can generate a large set of matches if a match mode which can generate more than one match is used.

Example:

```
match ss -fuzz 1 ClCCCl CCCl
match ss -fuzz 1 ClCCCl CCl
match ss -fuzz 1 ClCCCl ClCCl
```

The first example matches, since there is only a single unmatched substructure atom in the best mapping - one of the chlorine atoms- , but the second and third do not. The third example demonstrates that fail atoms are straightforwardly ignored, but their unmatched neighbors are not allowed to start new implicit fragments. The second chlorine atom in the substructure cannot match because it remains tethered to the main fragment, even if the excess carbon atom in the substructure is designated as the one allowed failure atom. Both example two and three will however match with a fuzz of 2.

-generalize

```
-generalize none:heteroatoms
```

In mode *heteroatoms*, all atoms except carbon and hydrogen are treated as a generic hetero atom type and match each other.

-heavyatoms

```
-heavyatoms any:all
```

If mode *all* is selected, all heavy atoms on the structure side must have been matched by substructure atoms for an overall hit.

-implicit_is_singlaro

```
-implicit_is_singlearo 0/1
```

If the flag is set, bonds which were not specified with an exact bond order (for example, in **SMARTS**) are handled as „single or aromatic“ query bonds, and not with their effective bond order or aromaticity status after decoding.

-includeflags_ss

```
-includeflags_ss flag_value
```

This option allows the selection substructure atoms for the match procedure which have one of potentially several bits set in the `A_FLAGS` property. The decoded flag values are used as a bit mask, and only those structure atoms which have one or more bits of the mask set are selected for matching. By default, all substructure atoms are used for matching. If both an inclusion flag set and exclusion flag set (option **`-excludeflags_ss`**) is specified, the inclusion list is processed first. From the remaining atoms, those which match the exclusion filter are removed.

-includeflags_st

```
-includeflags_st flag_value
```

This option allows the selection structure atoms for the match procedure which have one of potentially several bits set in the `A_FLAGS` property. The decoded flag values are used as a bit mask, and only those structure atoms which have one or more bits of the mask set are selected for matching. By default, all structure atoms are used for matching. If both an inclusion flag set and exclusion flag set (option **`-excludeflags_st`**) is specified, the inclusion list is processed first. From the remaining atoms, those which match the exclusion filter are removed.

-includelabels_ss

```
-include_ss labellist
-includelabels_ss label_list/aref_sequence
```

Select substructure atoms for use in matching. By default, all substructure atoms are used. If both an inclusion list and an exclusion list (option **`-exclude_ss`**) are specified, the inclusion list is processed first. From the remaining atoms, those which are also listed in the exclusion list are removed.

-includelabels_st

```
-include_st label_list
-includelabels_st label_list/aref_sequence
```

Select structure atoms for use in matching. By default, all structure atoms are used. If both an inclusion list and an exclusion list (option **`-exclude_st`**) are specified, the inclusion list is processed first. From the remaining atoms, those which are also listed in the exclusion list are removed.

-isotope

```
-isotope 0/1
```

This flag determines whether isotopic labeling is used for matching. By default, isotope label matching is not performed. If this flag is set, substructures with an isotope label must map onto a structure atom with the same isotope label. Even if this option is not set, explicit references to property `A_ISOTOPE` in atom query expressions are always evaluated and used to determine the match.

-kekule

```
-kekule none/odd/even/all
```

By default (value *none* or 0), the Kekulé bond order of aromatic bonds is not used for matching. A substructure aromatic bond matches a structure aromatic bond, regardless of whether their Kekulé bond orders are the same or not. If this flag is set to *all* (or 3), aromatic bonds are compared with the drawn bond order. This can be useful for example in order to find a sequence of atoms for perform a reaction transformation which allows a simple change of bond orders in the path without a complete rearrangement of the full π system. The modes *odd* and *even* are useful for controlled matching of certain heteroaromatic systems. In mode *odd* (or 1), the Kekulé bond order is used for all bonds which are only a member of aromatic rings with an odd number of atoms, while the order of bonds in even aromatic systems (including those which are simultaneously a member in an odd aromatic system) is disregarded. Mode *even* (or 2) is the complementary counterpart.

-limit

```
-limit n
```

Set the maximum number of reported substructure matches to *n*. Any additional matches which might be present are ignored. -*maxmatch* is an alias.

-mapanchor

```
-mapanchor 0/1
```

If this flag is set, an anchor set (see option **`-anchor`**) is automatically constructed from the values of the `A_MAPPING` properties on the substructure and structure. `A_MAPPING` is the default property to encode reaction mapping information. Both substructure and structure must possess valid `A_MAPPING` data, otherwise this option is ignored. If this condition is fulfilled, any substructure atom which has a non-negative mapping number[3] is anchored to its counterpart on the structure side with the same mapping number. If no such number is present, the command immediately returns zero matches and empty atom/bond/mol mapping variables, if these were specified. This option can be combined with a normal **`-anchor`** option. The anchor tables are cumulative in this case.

-maxopenlinks

```
-maxopenlinks n
```

Limit the number of open links of the substructure embedded in a match. Any continuation of the structure from the matched substructure into the unmatched parts except by hydrogen atoms is considered an open link. Example:

```
match ss -mode distinct CC CCCC
match ss -mode distinct -maxopenlinks 1 CC CCCC
```

The first example reports three matches, the second only two.

In the latter case, the substructure matches only at either end, because in case of a match in the middle of the C4 carbon chain there would be two continuation links. The **`-terminal`** option is equivalent to using this mode with an open link count of one.

For substructures which are larger than a single atom, the open link count is the number of matched structure atoms which have one or more open links. For single-atom substructures, the match count is directly the number of open links of the single matched structure atom.

This attribute is only checked for the root level of hierarchical substructure matches (e.g. when using Recursive **SMARTS** or a similar match mechanism, the match of the recursive fragments is not tested). Example:

```
match ss -maxopenlinks 1 {[C;$(*OC=O)]} $eh
```

This command checks that the matched carbon has only a single non-hydrogen continuation (necessarily into the ester group, e.g. it cannot match an ethyl or higher ester, only methyl), but this test does not apply to the ester fragment which is checked in a nested deeper level.

-mode

```
-mode first/all/canonic/distinctatoms/distinctbonds/distinctfirstatom/
distinctheavyatoms/distinctinneratoms/distinctmols/distinctssatoms/
nocommon/unique/bilateraldistinct/bestscore/bestscores/distinctscores/
bilateralunique/distinctfgatoms/nocommonfgatoms/nocommonheavyatoms/
conditionalnocommonfgatoms
```

This important option determines the substructure match mode. The default mode is *first*. In mode *first*, only the first, if any, match is returned, and any list variables used to capture the atom, bond or molecule maps use only a single level of nesting.

Mode *all* reports all (subject to a potentially set maximum number of results, see **`-limit`** option) all possible matches, which differ in at least one atom mapping relationship to any other reported match.

3. A zero or negative atom mapping value indicates an unmapped atom.

Example:

```
set nmatch [match ss -mode all CC CCC]
```

returns 4, because the C2 fragment can be embedded in forward and backward direction, and matched on either the first two or last two carbon atoms of the propane structure.

Mode *distinctatoms* only reports matches which map onto a different set of structure atoms. Example:

```
set nmatch [match ss -mode distinctatoms CC CCC]
```

returns 2 for the mapping of the substructure onto the first two, and the last two carbon atoms. The backward matches of the C2 fragment are not reported, because they do not cover a new set of atoms.

Mode *distinctheavyatoms* is similar to the *distinctatoms* mode, but only uses non-hydrogen substructure atoms for determining whether a match should be considered new and included.

Example:

```
set nmatch [match ss -mode distinctatoms {CC[#1]} CCC]
set nmatch [match ss -mode distinctheavyatoms {CC[#1]} CCC]
```

The first example reports an astonishing 10 matches, because the hydrogen atom can be mapped to either of the three terminal hydrogens, or two central hydrogens, and there are two distinct embeddings of the substructure C2 fragment. Mode *distintheavyatoms* reduces the number of reported hits to 2, because only the atom mappings of the two carbons in the substructure are considered. In many cases, hydrogens can be considered equivalent, and in these cases this mode comes in handy.

Mode *distinctinneratoms* is similar to *distinctheavyatoms*, but instead of ignoring all hydrogen atoms on the substructure when determining the novelty of a match, all terminal atoms (those with less than two bonds) are ignored in filtering new matches.

Mode *distinctfirstatom* is another mode with a modified view of what are distinct matches. This mode only looks at the structure atom matched by the first substructure atom.

Mode *distinctfgatoms* is another variant of *distinctheavyatoms* where in addition to hydrogen all carbon atoms which are not a functional group (having four bonded ligands, or being aromatic) are omitted in the match distinctiveness test.

Mode *distinctmols* requires that the substructure matches a different molecule in the structure ensemble in each accepted match.

Mode *distinctbonds* uses the set of matched structure bonds to determine whether a match is novel. For cage structures, there may be multiple matches of the same structure atoms, but matching different bond paths.

Mode *unique* is a stricter version of mode *distinctatoms*. Here, the matched atoms must additionally be topologically different, as determined by hash code property `A_HASH` (when matching without stereochemistry) or `A_STEREO_HASH` (in stereo match mode), or the isotope-aware variants thereof if isotope labels are checked.

Mode *bilateralunique* is a variant where pairs of matched substructure/structure atom labels are used to determine whether a label set is unique, not just the atom label set.

Mode *nocommon* only reports matches which do not share any common atoms. Example:

```
set nmatch [match ss -mode nocommon CC CCC]
```

returns only a single match, because the middle carbon atom in the structure is already matched by the first match. Unfortunately, the results of this match mode may depend on the numbering of atoms. If, by change, a C2 substructure fragment is first matched in the middle of a C4 chain, only a single match is found, but if it matches first at one of the ends, two matches are found, because the middle match, if found next, is discarded and then the other terminal match is accepted. The described effect is not a problem in all cases, depending on the nature of the substructure, but using this mode requires careful analysis.

Variants of this mode are *nocommonfgatoms* and *nocommonheavyatoms*. The first disallows matches which share a functional heavy atom (hetero atom or non-aromatic carbon with a multiple bond) with a previous match, while the second allows overlap of matched hydrogens, but not of other atoms. Like the plain *nocommon* mode, the results can be dependent on atom numbering. The mode *conditionalnocommonfgatoms* works like *nocommonfgatoms* if there are functionalized atoms in the match. If there are not, the mode is automatically and temporarily reset to *nocommonfgatoms* for checking this match for overlap with an existing match.

Mode *canonic* returns only a single match if one can be found, but uses atom hash codes to return a canonic match within the structure, regardless of atom order. The exact hash code properties used to determine the canonic match mirrors whether the match checks for stereochemistry and/or isotopes, as in mode *unique*.

Modes *distinctssatoms* and *bilateraldistinctdistinct* are only useful in contexts where only a part the substructure may be matched, for example when using the **`-fuzz`** option. Mode *distinctssatoms* is essentially the same as mode *distinctatoms*, only that the matched atoms on the substructure side are checked, not those on the structure side. Mode *bilateraldistinct* uses substructure atom/structure atom pairs instead of simple atom identities as criterion of distinctiveness.

Modes *bestscore*, *bestscores* and *distinctscores* can only be used if a match scoring mechanism has been configured. If that is the case, *bestscore* returns one of the matches with the best score, *bestscores* all matches which share the best score, and *distinctscores* a set of matches with all have different scores, omitting matches with duplicate scores.

-multihighlight

```
-multihighlight 0/1
```

If this option is set, and the options **`-atomhighlight`** and/or **`-bondhighlight`** are used, and more than one match is generated, the highlight atom and/or bond attributes are also set for the second and further matches, resulting in a highlight set which is the union of all matches. By default, only the first match is highlighted, even if more than one match is generated and reported.

-noaliphaticonaro

```
-noaliphaticonaro 0/1
```

If this flag is set, aliphatic bonds do not map on aromatic bonds. By default, and in the absence of other criteria determining the match of a bond, both single and double (but not triple or higher) aliphatic substructure bonds match aromatic structure bonds, and vice versa. If the flag is set, substructure bonds which are not marked aromatic, either by explicit attribute setting or indirectly by aromaticity analysis of the substructure fragment, do not match aromatic structure bonds. By default, this flag is not set. This option does not influence the processing of bond query expressions which explicitly reference properties such as `B_ORDER` or `B_ISAROMATIC`. These are evaluated in any case.

-noarobondfg

```
-noarobondbg 0/1
```

If this flag is set, aromatic bonds are not considered functional groups. This flag influences the interpretation of the *insulator* and *separator* pseudo-atoms, which are constructs used to separate functional groups in the match process. By default, aromatic bonds are considered part of a functional group.

-nochainonaro

```
-nochainonaro 0/1
```

If this flag is set, substructure chain bonds (acyclic bonds) do not match on aromatic structure bonds. By default, and if no options prohibiting this like **`-nosingleonaro`** or **`-nodoubleonaro`** are set, single and double chain bonds can match aromatic structure bonds.

-nochainonring

```
-nochainonring 0/1
```

If set, substructure chain atoms cannot match ring structure atoms.

-nodoubleonaro

```
-nodoubleonaro 0/1
```

If this flag is set, double bonds do not map on aromatic bonds. By default, and in the absence of other criteria determining the match of a bond, both single and double (but not triple or higher) aliphatic substructure bonds match aromatic structure bonds, and vice versa. If the flag is set, substructure double bonds which are not marked aromatic, either by explicit attribute setting or indirectly by aromaticity analysis of the substructure fragment, do not match aromatic structure bonds. By default, this flag is not set. This option does not influence the processing of bond query expressions which explicitly reference properties such as `B_ORDER` or `B_ISAROMATIC`. These are evaluated in any case.

-noheterofg

```
-noheterofg 0/1
```

If this flag is set, bonds to hetero atoms are not considered part of functional groups. This flag influences the interpretation of the *insulator* and *separator* pseudo-atoms, which are constructs used to separate functional groups in the match process. By default, bonds involving a hetero atom are considered part of a functional group.

-nomultibondfg

```
-nomultibondfg 0/1
```

If this flag is set, non-aromatic multiple bonds are not considered part of functional groups. This flag influences the interpretation of the *insulator* and *separator* pseudo-atoms, which are constructs used to separate functional groups in the match process. By default, non-aromatic multiple bonds are considered part of a functional group.

-nosingleonaro

```
-nosingleonaro 0/1
```

If this flag is set, single bonds do not map on aromatic bonds. By default, and in the absence of other criteria determining the match of a bond, both single and double (but not triple or higher) aliphatic substructure bonds match aromatic structure bonds, and vice versa. If the flag is set, substructure single bonds which are not marked aromatic, either by explicit attribute setting or indirectly by aromaticity analysis of the substructure fragment, do not match aromatic structure bonds. By default, this flag is not set. This option does not influence the processing of bond query expressions which explicitly reference properties such as `B_ORDER` or `B_ISAROMATIC`. These are evaluated in any case.

-nosuperatommultilink

```
-nosuperatommultilink 0/1
```

If set, substructure superatoms can only be matched in such a fashion that they have a single link to other matched parts of the structure.

-nosuperatomonh

```
-nosuperatomonh 0/1
```

If set, substructure superatoms cannot match structure hydrogen, even if their usual specification would allow it.

-omitrecursion

```
-omitrecursion 0/1
```

This options influences the way matches of recursive **SMARTS** fragments are reported. Internally, the first atom of a recursive fragment is represented by an *any* atom on the basic substructure. This placeholder atom and its mapped structure counterpart are reported in atom maps, and the bonds leading to the placeholder in bond maps. If this flag is set, the placeholder atom and its bonds are omitted from the maps.

Example:

```
match ss -omitrecursion 0 {C[$(OC)]} COC amap
match ss -omitrecursion 1 {C[$(OC)]} COC amap
```

In the first example, the atom map contains the pairs "{1 1} {2 2}", while in the second example only "{1 1}" is returned as atom map.

In any case, detailed mapping information about all the atoms and bonds of the recursive fragment is currently not directly available on the script level.

-openhcount

```
-openhcount 0/1
```

If this flag is set, all hydrogen counts are considered minimum values. If a matched structure atom possesses more hydrogens, the match still succeeds, even if the original comparison operator uses equality as criterion, provided that the compared property value is `A_HCOUNT`, the standard hydrogen count property, which is the default used by the various query syntax decoders of the toolkit. This option is unusual because it is also applied to comparisons in atom or bond query expressions. By default, this flag is not set.

Example:

```
match ss -openhcount 0 {[C;H2]} CC
match ss -openhcount 1 {[C;H2]} CC
```

The first example does not match, because both carbon atoms in the structure possess three hydrogen atoms, not two, while the second attempt succeeds. Note that the simple specification

```
match ss -openhcount x {[CH2]} CC
```

succeeds regardless of the setting of this flag. This is a side effect of the implicit expansion of **SMARTS** hydrogen atoms when they appear directly behind the atom symbol in the default **SMARTS** decoder mode, which is described in detail in the section about the handling of **SMILES** strings.

Alternatively, it is of course possible to either use standard **SMARTS** or-connected hydrogen count alternative values, or use the toolkit-specific range extensions, as in

```
match ss {[C;H2,H3]} CC
match ss {[CH{2-}]} CC
```

but in many cases this makes the query more complicated than necessary.

-overlap

```
-overlap none/any/nobonds/noembedding/distinctatoms/distinctmols/
         distinctpatterngroups/commonmol
```

This option controls how potential overlap of multiple substructure fragments on the target structure is handled. If the substructure contains only a single fragment, this option has no effect.

The default mode is *none*. In this mode, no overlap of substructure fragments on the target structure may occur. All fragments must be matched side by side, matching different structure parts.

Mode *distinctmols* is even more restrictive than mode *none*. In this mode, only one substructure fragment may be matched onto each structure fragment (i.e. molecule).

Mode *commonmol* is related. With that flag, all substructure fragments must match the same structure molecule in non-overlapping fashion.

In mode *any*, every substructure fragment is treated independently of any other substructure fragment. No information about any match by other fragments is used. Arbitrary overlap of the fragments on the target structure is allowed.

Mode *nobonds* allows the overlap of atoms, but not of bonds. In effect, multiple fragments may overlap at the edges, but not share any larger structure parts.

In mode *noembedding,* atoms and bonds may overlap, but no substructure fragment may be completely embedded into the matched structure part covered by another fragment, meaning that *at least one* of any pair of matching substructure fragments must match an atom which is not matched by the other fragment.

Mode *distinctatoms* is similar to mode *noembedding*, but in this mode any pair of matching substructure fragments at least one structure atom must be matched by *each* substructure fragment which is not matched by the other.

Mode *distinctpatterngroups* is rather special. It expects property `M_PATTERNGROUP` to be set for the substructure fragments. The property values are integers which separate the substructure fragments into groups with the same identifier. All fragments of a specific group value must match the same structure molecule without overlap. Substructure fragments with a different group value cannot match a structure molecule which is already matched by a substructure fragment with a different pattern group number, even if that would be possible without overlap. If `M_PATTERNGROUP` is not set, it is computed, where every substructure fragment is simply assigned a sequential number. This overlap mode is then equivalent to *distinctmols*.

Because internally bitsets are used to track the mapping of substructure fragments, the maximum number of fragments which may be used in any mode but *none* or *distinctmols* is 64. The *none* and *distinctmols* modes do not have a maximum fragment count.

-pionaro

```
-pionaro 0/1
```

If this flag is set, any bond between atoms which are part of a π system can match an aromatic bond. This option is intended to allow the reproduction of the behavior of the **DAYLIGHT** toolkit, which has a much broader idea about which ring systems are aromatic than the **CACTVS** toolkit in its default aromaticity mode. The **DAYLIGHT** toolkit recognizes rings with exocyclic keto groups, such as purines and pyrimidines, as aromatic, while this toolkit does not. If the option flag is set, aromatic fragments match on such systems. By default, the flag is not set. This option is outdated. It is recommended to use **DAYLIGHT** aromaticity for matching instead.

-queryatomexpansion

```
-queryatomexpansion 0/1
```

If this flag is set, query atoms which potentially cover multiple matched structure atoms are expanded to cover all matched atoms. Query atoms of this class include the **MDL** Beilstein types and the EliLilly Spinach types, but not superatoms (see -*superatomexpansion*). With a set flag, the matched atoms map potentially contains multiple entries for a single query atom. By default, such query atoms are only registered to match the first structure atom.

-remembercomplexmatches

```
-remembercomplexmatches 0/1
```

If set, any complex match expression that once a substructure atom is found to match an atom with all its atom match conditions, is assumed to match it again with a different overall substructure mapping without re-checking. This flag can accelerate matches, but must only be used if the general match of an atom does not depend on the match status of other substructure atoms.

-restrictsubstitution

If set, unmatched non-hydrogen portions of the structure can only be linked to substructure *any* atoms, or open valences. If open valences are used, the number of open valences linked to a substructure atom limits the number of unmatched heavy atom continuation atoms on the structure. Example: A substructure *phenyl-any* would match benzene (on a hydrogen substituent), toluene or other mono-substituted benzenes, but not higher-substituted compounds. A substructure *open*-C-*open* would not match tertiary or quaternary structure carbons.

-rotateterminals

```
-rotateterminals 0/1
```

If set, the 2D bond direction of matched structure-side terminal atoms (i.e. atoms with only a single bond) is adjusted in property `A_XY` to match that of the direction of the matched substructure-side bond. This option is for example useful to force the same orientation of hydrogens as in a template. Obviously, this option requires for useful results that the general orientation of the matched structure part is the same as that of the substructure pattern. This is usually enforced by combining this option with the *-align* option in the *rotate*, *redraw* or *besteffort* modes.

-spinachformula

```
-spinachformula formulaexpression
```

Define a formula match expression for *spinach*-class superatoms. The atom set matched by such an atom must match the formula expression, otherwise the match is rejected. This can for example be used to restrict the size and type of substituents. If no explicit spinach formula is set, it is taken from **`::cactvs(default_spinach_formula)`**, where it can be globally changed. Example:

```
match ss -spinachformula >=C1-2 {smarts:c1ccccc1[[CSPINACH]]} c1ccccc1C
match ss -spinachformula >=C1-2 {smarts:c1ccccc1[[CSPINACH]]} c1ccccc1CC
match ss -spinachformula >=C1-2 {smarts:c1ccccc1[[CSPINACH]]} c1ccccc1CCC
```

The first two expressions match. The carbon-spinach superatom gobbles up the entirety of the ring substituent atoms, and it has one or two carbons for the first two examples. The third example fails, because the substituent has too many carbons.

-stereo

```
-stereo none/absolute/relative
```

This option controls the global use of stereochemistry information of the substructure in the match process. By default, stereochemistry is ignored. If this flag is set, stereochemistry present in the substructure is checked against the stereochemical features in the structure. Stereo checks are performed on a pseudo-3D model of the compound and do not use simple descriptor values such as R and S.

If a stereo center in the substructure is unspecified, any stereochemistry, including unspecified stereochemistry, is allowed on the structure side in the matching atoms or bonds. If stereochemistry on an atom or bond of the substructure is specified, it must match the features found in the structure. Unspecified stereochemistry for the matched bond or center on the structure normally leads to a mismatch, except in case a *nostereook* flag has been set in `A_QUERY(`*flags*`)` or `B_QUERY(`*flags*`)` for the substructure atoms or bonds. Currently, the substructure match system handles stereochemistry of tetrahedral centers (including those which involve free electron pairs), cis/trans double bonds, allenes (both odd and even) and square planar geometries. Other geometries such as pentagonal bipyramids or octaeders are not yet supported.

With stereo match mode *absolute*, the pseudo-3D configuration of substructure and structure must match at all stereo centers and diastereomeric bonds specified in the substructure. The alternative mode *relative* allows the opposite configuration at stereo centers (but not bonds), provided that *all* matched stereo centers possess the opposite configuration. For example, an S,S-substructure would match both an S,S- and R,R-structure, but not the S,R or R,S-isomer. In effect, only stereo isomers are matched, but not diastereomers. The *relative* mode is obviously useful only when more than one stereo center needs to be matched.

Explicit atom stereo groups, such as the **MDL** stereo groups, override the global *absolute* or *relative* settings for the atoms involved.

Examples:

```
match ss -stereo none {[Cl,Br,I][C@H](CC)C} {C[C@H](CCC)Cl}
match ss -stereo absolute {[Cl,Br,I][C@H](CC)C} {C[C@H](CCC)Cl}
match ss -stereo absolute {[Cl,Br,I][C@H](CC)C} {C[C@@H](CCC)Cl}
match ss -stereo absolute {[Cl,Br,I][C@H](CC)C} {C[CH](CCC)Cl}
```

In this example set, the first line matches, because stereochemistry is ignored. The second line does not match, because the target structure represents the opposite stereo isomer. The third line does match, and the last line fails again because the substructure requested matching stereochemistry at a center for which no stereochemical information was available on the structure.

-strictexclusion

```
-strictexclusion 0/1
```

This is an expert option which controls how substructure fragments are handled which exclusively consist of atoms which bear the attribute that they should *not* be matched. By default, an attempt to match these fragments is performed *after* all other substructure fragments have been matched, and their matched structure parts are blocked. If at this point a match of any such fragment succeeds, the match is a failure. However, at this stage, structure parts which could match the exclusion fragment are potentially covered by other substructure fragments and thus protected, if the overlap mode disallows overlaps. If the flag is set, the check of these fragments is performed *before* the normal substructure fragments are processed. If a match occurs, the match process is immediately aborted.

-strictsmarts

```
-strictsmarts 0/1
```

If set, substructure argument specifications are decoded as strict **SMARTS** definitions. This means for example that the non-aromaticity of upper-case elements in **SMARTS** is enforced. Atoms for which aromaticity is not relevant need to be encoded with # notation, or as or-ed uppercase and lowercase element symbol pair. This flag only has an effect if the substructure is decoded within the **`match`** command. If the handle or reference of an existing ensemble is used as substructure specification, its internal representation and match behavior is not changed and was already defined by whatever decoder options were used when it was created.

-superatomexpansion

```
-superatomexpansion 0/1
```

If set, the match of a superatom is expanded to cover all structure atoms its definition encompasses. In the atom map, all these structure atoms are mapped to the same substructure superatom. By default, the match register only maps the first substructure/structure atom pair, and the other structure atoms are not explicitly registered as matched. Superatoms are not the same as expandable query atoms (see -*queryatomexpansion*).

-superatommapproperty

```
-superatommapproperty 0/1
```

If set, superatom matches are registered in property `A_SUPERATOM_SSMATCH`. In case there are multiple matches, each match generates its own property instance. The value for the matched structure atoms is the matching substructure superatom string copied from property `A_SUPERATOMSTRING`.

-tautomers

```
-tautomers none/basic/advanced
```

By default, bond orders and location of hydrogen atoms in the structure are fixed. A tautomer of a compound is considered a different chemical entity and does not match another tautomer. If the tautomer match mode is explicitly set to *none*, the match procedure continues to work in this style.

The alternative tautomer match modes *basic* and *advanced* introduce flexibility - at the cost of longer processing times, and a risk of obtaining matches which are surprising at first glance.

Examples:

```
match ss -tauto none {C=CO[H]} CC(=O)C
match ss -tauto none {CC=O} C=C(O)C
match ss -tauto basic {C=CO[H]} CC(=O)C
match ss -tauto basic {CC=O} C=C(O)C
```

The first two sample lines with the substructures of an enol and a keto group do not find a match with the structures of acetone and its keto form. The second pair of lines does find matches in both cases.

Atom and bond maps can be used with tautomeric matches, but the results can be surprising. The bond of a wandering hydrogen atom in the substructure is matched to the bond with the hydrogen in the original structure. However, since the substructure hydrogen atom may actually have been matched against a different virtual structure than the one passed to the match routine, the partner atoms of the bonds to the hydrogens in the substructure and structure may not have been mapped onto each other!

The difference between the *basic* and *advanced* modes is that the basic mode does not disturb aromatic systems, while the advanced mode considers forms which involve the conversion of aromatic systems into quinoids and vice versa, at the cost of extra processing time and less precisely defined matches.

The option can be shortened to -*tauto*.

-terminal

```
-terminal 1/0
```

This is another expert flag, and equivalent to the -*maxopenlinks* option with a link count of one. If it is set, a maximum of one bond, with the exclusion of bonds to hydrogen, may lead from the matched part of the structure to any non-hydrogen unmatched atoms. Essentially, the substructure is mapped into peripheral regions of the structure.

Example:

```
set nmatch [match ss -mode all CO C(O)C(O)C]
set nmatch [match ss -mode all -terminal 1 CO C(O)C(O)C]
```

In this example, the first line returns two matches, since the CO fragment can be matched onto both CO groups in the structure. The second line finds only a single match. The substructure cannot be matched onto the seconds CO group, because in that match the structure carbon atom has two unmatched non-hydrogen neighbors, one leading to the first CO group, and the other to the methyl group.

-timeout

```
-timeout nsecs
```

Set a time-out for the match operation. By default, or when a value of zero is given, the routine does not time out. If a time-out occurs, the match procedure is stopped. If any matches have been found so far, these are reported as results, without raising an error.

-transferstereo

```
-transferstereo none/atoms/bonds/both
```

If set to anything but *none*, which is the default, stereogenic atoms and/or bonds in the structure that are matched by substructure atoms or bonds with defined stereochemistry, and do not already possess their own stereochemistry descriptors, inherit stereochemistry from the substructure. This is done by setting properties `A_LABEL_STEREO` or `B_LABEL_STEREO` in such a fashion that the absolute configuration is the same as in the substructure. Depending on the atom and bond labeling of the structure vs. substructure, this is not necessarily the same descriptor value. In order for such a match to succeed, missing atom or bond stereochemistry on the structure side needs to be allowed (see -*allowmissingstereo* option).

-useatomtree

```
-useatomtree 0/1
```

This flag is set by default, but may be reset with this option. If the flag is set, atom query expression trees present in property `A_QUERY(`*query*`)` are evaluated and used to determine match possibilities. If this flag is not set, query trees are ignored and only the flat atom match attribute set is used.

-useatomtype

```
-useatomtype 0/1
```

This flag is set by default. If it is reset, atom type information is not checked in matching. *-atomtype* is an alias.

-usebondorder

```
-usebondorder 0/1/2
```

This flag determines whether the bond orders of substructure and structure bonds outside aromatic systems is used for determining a match. By default this flag is ***set*** to 1, but may be disabled with this option. This option affects only the basic bond match. Bond match query expressions which explicitly or implicitly refer to property **`B_ORDER`** always use their comparison results to determine matches. In rarely used mode 2, bond order matching is only used for terminal structure bonds (i.e. those which contain an atom which participates only in a single bond). *-bondorder* is an alias.

-usebondtree

```
-usebondtree 0/1
```

This flag is set by default, but may be reset with this option. If the flag is set, bond query expression trees present in property `B_QUERY(`*query*`)` are evaluated and used to determine match possibilities. If this flag is not set, query trees are ignored and only the flat bond match attribute set is used.

-varbondglobal

```
-varbondglobal maxdelta
```

If this option is used, the global use of approximated fractional bond orders for coordinate compound hypergraph matching is enabled for bonds with explicit approximated bond order request values stored in property `B_QUERY(`*varbo*`)`. The *maxdelta* parameter is the maximum allowed average deviation of the matched structure bonds (with fractional order in `B_ORDER_ESTIMATE`) vs. the substructure bonds that have a specified value in `B_QUERY(`*varbo*`)`.

-varbondlocal

```
-varbondlocal maxdelta
```

If this option is used, the use of approximated fractional bond orders for coordinate compound hypergraph matching is enabled for bonds with explicit approximated bond order request values stored in property `B_QUERY(`*varbo*`)`. The *maxdelta* parameter is the maximum allowed individual deviation of the fractional query bond orders in `B_QUERY(`*varbo*`)` from the structure-side fractional bond order values of matched bonds stored in property `B_ORDER_ESTIMATE`.

-wedge

```
-wedge 0/1
```

If this flag is set, matching bonds on the substructure and structure sides must possess identical wedge attributes (both wedge tip location and up or down direction). This option should only be used under very specific circumstances. It is **not** a replacement for stereo center matching, since wedges can be placed onto different bonds around a stereo center, and still represent the same stereo isomer.

Maximum Common Substructure Matching

Maximum common substructures can be found with the *mcss* mode. Algorithmically, this works by gradually increasing the *-fuzz* parameter until a match is found.

Example:

```
set eh1 [ens create CCC(=O)C]
set eh2 [ens create ClCCC]
match mcss -excludeh all $eh1 $eh2 amap
echo $amap
match mcss -mode all -excludeh all $eh1 $eh2 amap
echo $amap
```

The first command finds one substructure (first three carbons of the first structure mapping on atoms 2-4 of the second structure. The second command reports also alternative solutions, all with three carbons. Excluding the hydrogens from the match is not required, though it accelerates the process.

All match options can be used for this command variant - including, for example, the *-align* variants, which result in the structures being aligned by a common core.

Reaction Substructure Matching

Reaction substructure matching is very similar to simple substructure matching. The substructure and reaction arguments are required to represent reactions, not ensembles. All standard configuration options can be used. Configured match options are implicitly applied both to the reagent and product side substructure matches.

In the normal case, both the reaction pattern and full reaction should be mapped, i.e. both sides of the reactions should possess property `A_MAPPING` which contains the label of the atom on the other side corresponding to the atom bearing the property value, or 0 for atoms which are only present on one side. This property is automatically set on the reagent and product ensembles when reading reaction files with mapping information, or decoding a **Reaction SMILES** with map IDs. The reaction match algorithm automatically uses the mapping to construct an anchor table for the second stage matching. This second-stage anchor table is independent of an anchor table specified as command argument - a custom anchor table will be used only in the first-stage match if it is set.

Matching happens in two levels of recursion. First, a normal substructure match is attempted between the substructure ensemble with the *reagent* `E_REACTION_ROLE` and the corresponding reagent ensemble of the full reaction. If that succeeds, the query ensemble with the *product* `E_REACTION_ROLE` is matched in a recursive call against the corresponding full reaction product ensemble. If both the query and the full reaction have `A_MAPPING`, an additional constraint is enforced that the product query atoms with a non-zero `A_MAPPING` can only match the full product atoms which have the same `A_MAPPING` as the full reagent atoms which were matched by the reagent query atoms with the same `A_MAPPING` as the current product query atom.

This mechanism ties the two query sides together and makes a full reaction query different from a query which only looks for the independent presence of the query patterns in both sides of the reaction. It is possible to run a reaction query without the implicit `A_MAPPING` tie by omitting the mapping, either on the query or the full reaction. In that case, the effect is an independent match of the patterns on both sides.

`A_MAPPING` is computable on the full reaction or reaction query pattern, but the current implementation is comparatively underoptimized, and for more complex query patterns likely to report spurious results. In general, reaction mapping has a very problematic algorithmic complexity power law (NP problem). The toolkit implementation only works reliably and with usable speed for small molecules and simple query patterns. The reaction match command does not attempt to compute the property (neither on the pattern nor on the full reaction side) if it is not present.

If match variables are configured to capture detailed information on the match result, reaction substructure matches utilize two independent sets - one for the reagent-side match and one for the product-side match. The format of results stored therein is the same as for normal substructure matches. It is possible to configure any subset of these variables and to ignore other information.

Examples:

```
set xh1 [reaction create c1ccccc1C(=O)C>>c1ccccc1C(O)C]
```

```
set xh2 [reaction create {[c:1]1[c:2][c:3][c:4][c:5][c:6]1[C:7](=[O:8])[C:9]
>>[c:1]1[c:2][c:3][c:4][c:5][c:6]1[C:7]([O:8])[C:9]}]
```

Two identical reactions (reduction of phenylketone), without and with atom mapping. The reactions may of course be created from other formats, such as **MDL RXN** files, **KEGG** reaction IDs, or **CHEMDRAW** files.

```
set xss1 [reaction create {[c:1]-[C:2]=[O:3]>>[c:1]-[C:2]-[O:3]}]
echo [match rss $xss1 $xh1] (1)
echo [match rss $xss1 $xh2] (1)
```

Both match - the first reaction because no reaction mapping is present on the reaction and so this is a simple double substructure check - while in the second case the atoms match as indicates by the mapping, and the bonds change as required.

```
set xss2 [reaction create {[c]-[C]=[O]>>[c]-[C]-[O]}]
echo [match rss $xss2 $xh1] (1)
echo [match rss $xss2 $xh2] (1)
```

Both match because there is no mapping on the query, and this again is a simple double substructure match.

```
set xss3 [reaction create {[c]-[C]=[O]>>[c]-[C]-[c]}]
echo [match rss $xss3 $xh1] (1)
echo [match rss $xss3 $xh2] (1)
```

Once more: No mapping, simple double substructure match - matching the right-side pattern substructure on a different location unrelated to the left side is no error.

```
set xss4 [reaction create {[c:1]-[C:2]=[O:3]>>[c:1](-[C:2])-[c:3]}]
echo [match rss $xss4 $xh1] (1)
echo [match rss $xss4 $xh2] (0)
```

But this does not work any longer with mapping present both on the pattern and the full reaction. The third query atom does not get converted to a carbon in the reaction. Having mapping on both the full reaction and the query is the standard case for reaction substructure matching.

```
set xss5 [reaction create {[c:1]-[C:2]-[O:3]>>[c:1]-[C:2]=[O:3]}]
echo [match rss $xss5 $xh1] (0)
echo [match rss $xss5 $xh2] (0)
```

Here the pattern is for the reverse reaction, not the forward direction. Since the direction of the reaction is encoded in the reaction object, this does not work. See the **`reaction reverse`** command for a possibility to reverse the encoding of a reaction.

Commands for Structure-unrelated Objects

The **Cactvs** system maintains a set of objects which are not directly connected with chemical entities. These objects do not participate in the standard property scheme. Rather, they are configurable only through a set of fixed attributes, which is not extensible.

The following commands are currently used to manage non-chemical objects:

- **`cmdx`**
  Tcl extensions similar to standard **Tcl** packages, but with more metadata
- **`dbase`**
  perform interaction with database servers
- **`dbx`**
  manage database interface modules
- **`factory`**
  manage chemical data processing workspaces
- **`filex`**
  manage handler modules for chemical structure and reaction file formats
- **`filter`**
  manage property filters
- **`lhasa`**
  manage **Lhasa/CHMTRN**-style chemical reaction rule processing
- **`knode`**
  manage **KNIME** node objects.
- **`keyx`**
  handler modules for associating property data with keys from database tables
- **`netx`**
  manage handler modules for network file formats
- **`prop`**
  manage property definitions
- **`repx`**
  manage interface modules for handling alternative representations of chemical objects
- **`soap`**
  manage **SOAP** communication and **XML** parsing
- **`station`**
  manage pipelined chemical structure data processor objects
- **`tablex`**
  manage handler modules for table file formats
- **`typex`**
  manage handler modules for extended data types

The cmdx Command

The cmdx command is used to manage **Tcl** command extensions. **Cactvs Tcl** command extensions are an upward-compatible extension to standard **Tcl** packages. It is possible to load them by means of the standard **Tcl** commands `load` or `package require`, but if loaded in that fashion, the additional metadata and attributes are not accessible.

This command is not supported in the **Python** interface.

The following subcommands exist:

cmdx defined

```
cmdx defined cmd
```

Check whether a module for the specified command is either already loaded, or available. If a module can be found, and it not yet loaded, it is automatically loaded.

The return value is a boolean status code. No error is generated when the command cannot be resolved to a module.

cmdx exists

```
cmdx exists cmd
```

Check whether a module for the specified command is already loaded. No attempt is made to auto-load a module if it is not already loaded.

The return value is a boolean status code. No error is generated when the command cannot be resolved to a module.

cmdx get

```
cmdx get cmd attribute
cmdx get missed
```

Query the value of an attribute of the extension module. Tcl command extension modules are static. There are no `cmdx create` or `cmdx set` commands to define command extensions in a script, or to modify the attribute set of a module. The following attributes can be queried:

- *address_city*
  The city part of the author contact address.
- *address_country*
  The country part of the author contact address, following the ISO3166 standard.
- *address_state*
  The state part of the author contact address. Empty if not applicable.
- *address_street*
  The street address part of the author contact address. Includes floor, house number, etc.
- *address_zip*
  The **ZIP** code or other applicable postal code of the author contact address.
- *affiliation*
  The institution the author works for.

- *affiliationduns*
  The **DUNS** registration ID of the affiliated institution. This is primarily useful for US government projects.
- *affiliationurl*
  The **URL** of the affiliated institution.
- *author*
  The author of a command extension, as free text.
- *authorurl*
  A **URL** with information on the author, or an empty string if unset.
- *category*
  A category string to be used if the module is stored in a repository.
- *classuuid*
  The base class **UUID** of this module.
- *comment*
  A free-form comment.
- *date*
  The date of the last change of the module source code.
- *doi*
  A digital object identifier for the module, if defined.
- *email*
  An email address of the author to facilitate contact.
- *infourl*
  A **URL** with information on the module, or an empty string if unset.
- *keywords*
  A list of keywords associated with the module.
- *license*
  The license class associated with this module.
- *licenseurl*
  A **URL** with details about the module license.
- *literature*
  A free-form literature reference.
- *name*
  The name of the command.
- *objectfile*
  The full path name of the loaded object file.
- *orcid*
  The **ORCID** code of the author (see www.orcid.org).
- *path*
  The repository path for displaying hierarchical repository trees. This attribute is independent of any file system paths.
- *phone*
  A contact phone number of the author.

- *references*
  Cross references of the module. This is a nested list of class **UUID**s and reference type tags.
- *regid*
  In case this is a registered module, its official ID. Unregistered modules report zero.
- *sourcefile*
  The name of the source file of the module, if available.
- *version*
  Version information. This is a string in a 1.2.3 (or shortened) style.
- *versionuuid*
  The version **UUID** associated with this module.

The second variant with the *missed* special command name lists all commands which could not be resolved via *load* subcommands - which could have occurred implicitly via the standard (but extended to include **`cmdx load`** attempts) **Tcl** external command executable lookup function. The **`cmdx`** command maintains a cache of such misses in order to avoid searching the command extension path repeatedly, which can generate significant network traffic if this happens frequently. On the other hand, it also means that it is not possible to supply a **`cmdx`** module late during a script run when a command load attempt has already failed.

cmdx list

```
cmdx list ?pattern?
```

Get a list of all currently loaded **Cactvs Tcl** command extensions. This list does not include standard **Tcl** extensions. If desired, the list can be filtered by a string match pattern.

cmdx load

```
cmdx load cmd ?objectfile?
```

Explicitly load a command extension module. If the module is already loaded, the current version is unloaded first. If no specific object file (a shared library on Unix/Linux, a DLL on Windows, a bundle file for MacOSX) is specified, the standard name of the module file is automatically generated from the data type name, and then the file searched in the directories in the data type handler module path. The module path can be customized in the control variable *::cactvs(cmdxpath)*.

After loading, the *cmd* command is available in the interpreter which executed the script command. It is essentially indistinguishable from built-in commands. Command extensions are global and automatically available both in **Tcl** slave interpreters (for scripted property computations) and forked threading **Tcl** interpreters, provided that these interpreters are created after the extension has been loaded into the main interpreter. Pre-existing interpreters of these types do not retroactively obtain access to the command when it is loaded into the main interpreter. Loading extensions directly into slave interpreters or thread interpreters is not possible, because these do not support the **`cmdx`** command. Because of these complications, command extensions should preferably be loaded at the very beginning of a script, before threads are forked or property slave interpreters are instantiated.

The return value of the command is the slot in the command extension table the module has been loaded into.

cmdx subcommands

```
cmdx subcommands
```

Return a list of all subcommands of the **cmdx** command.

cmdx unload

```
cmdx unload ?cmd?..
```

Unload one or more **Tcl** extension modules. It is an error to specify the name of a module which is not loaded. It is not advisable to unload a command extension which has already been exported to slave or thread interpreters, because this can lead to crashes if these interpreters attempt to use the extension after it has been unloaded.

The dbase Command

The **`dbase`** command is used to interact with database servers. While the command provides a generic, database-independent set of features, the actual interaction is performed via loadable database driver modules. These are either loaded explicitly (via the **`dbx load`** command) or automatically be referring a known database type name.

The **`dbase`** command provides the following subcommands:

dbase close

```
dbase close all
dbase close ?dbhandle?...
db.close()
Dbase.Close(?dbref/dbhandle?,...)
Dbase.Close("all")
```

Close open database connections held by the specified database access objects and destroy the connection manager objects. The handles passed to this command are no longer valid after the command has been executed.

The magic handle name *all* can be used to close all currently opened database connections.

Example:

```
dbase close all
```

The return value is the number of closed database objects. For the sake of consistency with other object commands, the command **`dbase delete`** is an alias to this command. Both commands do not delete any database on the server.

dbase columnquery

```
dbase columnquery dbhandle sqlstatement ?tablehandle|new?
db.columnquery(query=,?table=?)
```

This command is a variant of the **`dbase query`** command. By default, the command returns a nested list of rows and columns. This command only returns the first result column, if any are produced, and omits the outer nesting level. This can make the processing of results easier. Example:

```
set smileslist [dbase colquery $dbhandle "select smiles from moltable"]
```

If a table handle is specified as target, or a table was created, the return value is the table handle or reference.

The command can also be accessed under the shortened name *colquery.*

dbase connect

```
dbase connect dbhandle
db.connect()
```

Establish a connection to the database server. An error is thrown if the connection does not succeed. This statement is primarily useful to verify the correctness of the attributes set by means of **`dbase create`** and **`dbase set`** commands. For the execution of database commands it is not required. In case a database connection was not yet established when communication with the server is required, an attempt to open the connection is made automatically.

If the *dsn* (data source name) attribute has been set, it has precedence over the connection parameters defined by the database *host, port, database, user* etc. attributes.

The command returns the original database connection handle or reference.

dbase create

```
dbase create ?attribute value?...
dbase create dict
Dbase(?attribute,value?,...)
Dbase(dict)
Dbase.Create(?attribute,value?,...)
Dbase.Create(dict)
```

Create a new database access object. Any number of database access objects can be in existence at the same time, and be connected to the same or different databases, potentially using different database drivers. If no attributes are specified, a default database object is created. Some of the default values can be modified via elements in the *::cactvs()* control array:

- *default_database*
  The name of the database. The default is the user name.
- *default_database_host*
  The database host. The default is *localhost*.
- *default_database_options*
  An option string for drivers which support this. Empty by default.
- *default_database_password*
  The database password, if required. Empty by default.
- *default_database_type*
  The database type. Set to *mysql* by default. The default port is automatically set to the default communication port of the database type (3306 for the *mysql* case).
- *default_database_user*
  The user name used to connect to the database. By default, this is the same as the user name.

The attributes which may be set by this command the same as in the **`dbase set`** command and explained there. The return value of the command is the database object handle, which is used to identify the object in all further operations.

Example:

```
dbase create dbtype mysql database samples host db3 user beaker password muppet
```

Note that this command only sets up the database interface configuration, but does not immediately open a connection. A connection to the database is only opened the first time there is a need to communication with the database server, of the **`dbase connect`** command is executed. Until then, it is for example possible to set additional parameters via the **`dbase set`** command which are used when the connection is finally established.

`dbase open` is an alias.

dbase disconnect

```
dbase disconnect dbhandle
db.disconnect()
```

Close the connection to the database established via a **`dbase connect`** command or implicitly by data retrieval commands. The interface object remains valid and can, potentially after a change of attributes, reconnect to a database.

In case the interface object was not yet connected, the command does nothing.

The command returns the original database connection handle or reference.

dbase dup

```
dbase dup dbhandle
db.dup()
```

Duplicate the attributes of an existing database interface object into a new object. The return value is the handle or reference of the new interface object.

This command only copies the configuration options, but does not inherit the database connection, or any related state information. The new interface object is in the same state as if it were created via a **`dbase create`** statement with a complete set of attribute and value pairs.

dbase exec

```
dbase exec dbhandle sqlstatement ?tablehandle?
db.exec(query=,?table=?)
```

This command is a variant of the **`dbase query`** command. The difference is that any returned results are discarded and the return value is the current boolean value of the *iserror* database connection attribute. In case of severe errors. a **TCL** or **PYTHON** error is generated.

dbase exists

```
dbase exists dbhandle ?database?
db.exists(?database=?)
Dbase.Exists(connection=,?database=?)
```

Test whether the specified database is visible via the current connection or not. If no database name is specified, the current value of the *database* attribute of the interface object is used for the test. The return value is the boolean test result. If the connection cannot be established, an error is generated.

dbase flush

```
dbase flush ?dbhandle?
db.flush()
Dbase.Flush()
```

This command flushes the internal database caches globally or only those associated with the connection. The toolkit remembers certain information, such as database and table names, or database column types, in order to accelerate processing. In case the database content was modified by deleting, adding or altering tables, or full databases have been deleted, renamed, or created, it is advisable to use this command to make sure that the cached information does not become outdated and a source of error.

In circumstances where a database my be accessed via more than one connection, it is best to flush the caches globally, or on all connections which operate on that database if the exact set of affected interface objects is known. Extraneous flushing of the caches does not change any valid results, but can lead to performance degradation.

dbase get

```
dbase get dbhandle attribute
db.get(attribute)
db.attribute
db[attribute]
```

Read a database interface object attribute. The list of attributes is explained in the paragraph on the **`dbase set`** subcommand.

Example:

```
set id [dbase get $dbhandle insertid]
```

dbase itemquery

```
dbase itemquery dbhandle sqlstatement ?tablehandle|new?
db.itemquery(query=,?table=?
```

This command is a variant of the **`dbase query`** command. By default, the query command returns a nested list of rows and columns. This command only returns the first result item, from the first row and first column, if any are returned, and omits the standard two layers of list wrappers. This can make the processing of results easier. Example:

```
set size [dbase itemquery $dbhandle "select count(*) from moltable"]
```

If a table handle is specified as target, or a table was created, the return value is the table handle or reference.

dbase list

```
dbase list ?pattern?
Dbase.List(?pattern=?)
```

Return a list of all currently defined database connector handles.

dbase query

```
dbase query dbhandle sqlstatement ?tablehandle|new?
db.query(query=,?table=?)
```

Execute an **SQL** statement on the database server. The allowed **SQL** syntax is dependent on the capabilities of the connected server.

The default return value is, in the absence of the optional table handle argument, a nested list of rows and columns, with the rows as the outer list level. The maximum number of returned rows can be controlled by means of the *maxrows* interface object attribute. If table column data type information is available, the internally used **Tcl** or **Python** result objects are matched to the column type for increased performance. Otherwise, the returned items are strings.

This command is not limited to the execution of **SQL `select`** statements. Any supported statement can be executed. In case it does not return a result tuples, an empty set is returned.

In case the optional target table handle argument is supplied, the result data is directly stored in the specified table object. When the argument is present and explicitly set to an empty string, or the magic value *new*, a new table is created, which is automatically destroyed in case the command fails. An attempt is made to map existing table columns to the names of the database query result columns. In case no matching table object columns can be found, they are added automatically to the right with suitable data types, names, precision, width, and so on. Existing table columns which do not receive data from the result set are set to **NULL** values in the new rows. Existing table object rows are not deleted when the command is run. The retrieved rows from the database result set are appended. In this mode, the return value of the command is the table handle.

Example:

```
set th [table create]
dbase query $dbhandle \
   "select smiles,name from moltable where logp between 5.0 and 6.0" $th
set nrows [table get $th nrows]
```

For convenient handling of single-row, single-column and single-item queries, there are the command variants **rowquery**, **columnquery** and **itemquery** which omit one or two levels of list nesting, and only retrieve a single row, column or item, ignoring any additional data returned by the database.

dbase matrixquery is an alias for this command which emphasizes the relationship of this command to the **rowquery**, **columnquery** and **itemquery** command variants.

Earlier versions of the toolkit returned not just the table handle, but a list of the handle and the row and column counts if the query result data destination was a new or existing table. This was changed in version 3.4.6.18 because it was not found useful for practical scripting.

dbase queryloop

```
dbase queryloop dbhandle sqlstatement varname ?offset? ?maxrows? body
```

Execute a loop over a result set returned by the **SQL** query. For every result row, the row values are stored in a list variable and then the body code is executed. The row variable is locally visible from within the body code. Within the body code, standard loop control statements like **continue** and **break** can be used in the normal fashion. Optionally, an offset into the result set may be specified (default 0) and a maximum number of iterations (default -1, no limit).

Example:

```
dbase queryloop $dh "select * from molecules" row {
   puts $row
}
```

dbase read

```
dbase read dbhandle ?sqlfile?...
db.read(?sqlfile?,...)
```

Open the specified file(s) and extract **SQL** commands one by one. Every **SQL** command is then processed by the equivalent of a **dbase exec** statement. Processing stops if an error is encountered.

Individual **SQL** statements in the file may span multiple lines, and the file may contain empty lines or **SQL**-compatible comments, which are skipped. Extended $-quoting (as in nested **$a$** vs. **$b$** sections) is currently not handled as explicit quoting which temporarily suspends the detection of the end of a statement. Only standard single-character quoting and $$ quoting (as well as their escaping within quoted sections) is explicitly parsed, so extended quoting must only be used within standard-quoted sections if their contents could be misconstrued as containing statement ends.

The command returns the original database connection handle or reference.

dbase ref

```
Dbase.Ref(dbhandle)
```

Python-only method to get a reference of the connector from its handle.

dbase rowquery

```
dbase rowquery dbhandle sqlstatement ?tablehandle|new?
db.rowquery(query=,?table=?)
```

This command is a variant of the **`dbase query`** command. By default, that command returns a nested list of rows and columns. This command only returns the first result row, if any are received, and omits the column nesting level. This can make the processing of results easier. Example:

```
dbase rowquery $dbhandle "select smiles,weight from moltable where cas='71-43-2'"
```

If a table handle is specified as target, or a table was created, the return value is the table handle or reference.

dbase set

```
dbase set dbhandle ?attribute value?
dbase set dbhandle dictionary
db.set(?attribute,value?,...)
db.set(dict)
db.attribute = value
db[attribute] = value
```

Set one or more attributes of the database interface object. Not all attribute changes have an effect after the database connection has been established. Generally, attribute changes which would necessitate the closing and re-opening of the database connection to a different host, or with different access credentials, are ignored after the connection has become active. If such a change is needed, a **`dbase reset`** command should be issued, or the current interface object should be discarded and a new one created.

Some attributes are read-only. They are listed here nevertheless, because the **`dbase get`** command refers to this section.

The following attributes are currently supported:

- *appname*
  The application name, which may be of interest to the database interface driver. By default it is set to an empty string, and most database interfaces ignore this attribute.
- *blocksize*
  The transmission block size used for database communication. The default, and in anything by exceptional circumstances the only value ever needed, is -1, which means to use the driver-specific default. This attribute is only of interest to database drivers which support this concept, for example the **TDS** driver for connecting to **MS SQL Server** databases.

- *clientinfo*
  The client-info string provided by the database interface library, if it has such a feature. This attribute is read-only.

- *clientname*
  The client name, which may be of interest to the database interface driver. By default it is set to an empty string, and most database interfaces ignore this attribute.

- *connected*
  A boolean read-only value indicating whether the connection to the database server has been established or not.

- *connectionstring*
  For database interface libraries which support this concept, a connection string encapsulates all required access information into a single string. If it is not set, an attempt is made to construct a suitable connection string from the basic *host, port, user, password* etc. attributes if the interface requires it. An explicitly set connection string is reset if the *host, port* etc. attributes of the interface object are changed.

- *database*
  The name of the current database. If it is changed between calls to the database server, interface commands are automatically issued to synchronize the name of the current database before the next **SQL** command is executed. The name of the initial database is copied from the value of the control variable `::cactvs(default_database)` if it has not been set explicitly by means of `dbase create` or `dbase set` statements. In case of the Oracle interface, the value is a service/schema name from the Oracle *tnsnames.ora* configuration file, since Oracle does not have a simple concept of a database.

  This attribute is not the same as *databases* (plural).

- *databases*
  This is a read-only attribute. It returns a list of the databases which are visible via the current connection. In case of Oracle, the list only contains the current service/schema name. No attempt is made to parse the *tnsnames.ora* configuration file.

  This attribute is not the same as *database* (singular).

- *dbtype*
  The type of the database to connect to. The default is copied from the value of the control variable `::cactvs(default_database_type)`. If a driver for the specified database type is not yet loaded, an attempt is made to auto-load it. The possible values for this attribute depend on the set of available database interface modules. Examples are *mysql*, *odbc*, *tds* (to connect to **MS SQL SERVER**), *postgres* and *oracle*.

- *description*
  A free-form string which can be used to add a descriptive text. The default is an empty string.

- *domain*
  The database domain, if the database driver uses this concept. The default is an empty string.

- *driver*
  The name of the driver module for meta-interfaces such as **ODBC**.

- *dsn*
  For the **ODBC** driver, the data source name. In a properly configured **ODBC** environment, this name is used to look up other required connection data, such as host name and applicable driver module, via a single identifier. In a **CACTVS** installation, the database source name definition file resides in the *odbc* subdirectory of the data directory (control variable `::cactvs(data_directory)`). The **DSN** string is reset when the *host, port, user, password* or *database* attributes are modified.

- *host*
  The name of the database host. The default database host is copied from the value of the control variable `::cactvs(default_database_host)`.

- *hostinfo*
  The host-into string provided by the database interface library, if it has such a feature. This attribute is read-only.

- *insertid*
  The last automatically generated id received from the database, for example from inserts into database tables with auto-increment columns. This is not supported on all database interfaces - for example, **ORACLE** requires you to query sequence data explicitly. The value is read-only.

- *iserror*
  A read-only boolean flag to indicate that the last database operation resulted in an error or warning. Not all database operation problems are translated into **TCL** script language errors. In order to become aware of non-critical problems, is flag should be checked in robust applications.

- *language*
  The language used for database interface messages, if the interface library supports this.

- *lastquery*
  The last **SQL** query executed on the database. This attribute is automatically updated when `dbase query/exec/rowquery/columnquery/itemquery` commands are run, but there are also some conditions when implicitly assembled **SQL** commands are run which are also tracked here. Setting this attribute is possible, but not very useful in normal script environments.

- *logfile*
  The name of a file to use logging for this connection, if the database interface library supports such a feature.

- *maxrows*
  The maximum number of rows read in a result set from a query. If the database interface supports this function, this is enforced on the server side. In any case, the `dbase query` command variants honor this attribute on the client side. If this attribute is set to zero or a negative value (the default), no row limit applies.

- *message*
  The last message string received from the database. This attribute is read-only. *msg* is an alias for this attribute.

- *null*
  The string value used to represent **NULL** database values in **TCL** return results. By default, it is am empty string, but this is indistinguishable from zero-length strings and therefore it is sometimes useful to change it to something else, such as the string "NULL". For **PYTHON**, **None** values are always used.

- *options*
  An option string which is passed to the database driver, if it supports this concept. The default option string is copied from the value of the control variable **::cactvs(default_database_options)**.

- *password*
  The password sent to the database server as part of the credentials. If no password is needed, use an empty string. The default password is copied from the value of the control variable **::cactvs(default_database_password)**.

- *port*
  The communication port used to talk to the database server. When the database type is set, it is automatically set to the default port for that database (i.e. 3306 for *mysql*). If a custom port is used, you therefore need to set it after the database type has been specified.

- *protocolinfo*
  The protocol-info string provided by the database interface library, if it has such a feature. This attribute is read-only.

- *protocolversion*
  The protocol version used by the interface library, if it can supply this information. This attribute is read-only.

- *serverinfo*
  The server-info string provided by the database interface library, if it has such a feature. This attribute is read-only.

- *socket*
  A the name of an Unix domain named socket, if it is used by the database interface. This attribute should be considered read-only except for database wizards who need to cope with non-standard database server installation settings on the local host.

- *table*
  The current table of interest. This is used for example in constructing database **URL**s. It has no direct influence on **SQL** query or command execution, and is not automatically updated when running **SQL** commands.

- *tables*
  This is a read-only attribute. It returns a list of the database tables visible in the current database via the current connection.

- *textsize*
  The maximum text block size used in database communication. The default, and in anything by exceptional circumstances the only value ever needed, is -1, which means to use the driver-specific default. This attribute is only of interest to database drivers which support this concept, for example the **TDS** driver for connecting to **MS SQL SERVER** databases.

- *trace*
  A boolean flag which is used to enable or disable **SQL** statement execution tracing. By default tracing is off. The output is written to the file specified in the *tracefile* attribute.
- *tracefile*
  The name of a file to write database access trace information to. The default is *sql.log*.
- *timeout*
  The time-out value in seconds. If a database response it not received in time, an error results. If the value is set to 0 (the default), no time-out is active.
- *url*
  A database **URL** constructed from the currently set database object attributes (host, port where applicable, database/schema/service name, user, password, table of interest, query). This argument is read-only.
- *user*
  The user name used to provide credentials to the database server. By default, it is the database user set in ::**`cactvs(default_database_user)`**, which again by default is the same as the login user name.

dbase tablequery

```
dbase tablequery dbhandle sqlstatement
db.tablequery(query)
```

This is a short form of **`dbase query`** with the optional table argument set to *#new*. The command returns the handle or reference of a new table with the result data, and the query expects multiple rows and columns of data.

dbase subcommands

```
dbase subcommands
dir(Dbase)
```

This command returns a list of all the defined subcommands of the **`dbase`** command.

The dbx Command

The **`dbx`** command is used to manage database driver extensions. The command has the following subcommands:

dbx defined

```
dbx defined dbtype
Dbx.Defined(dbtype)
```

Check whether a driver module for the specified database type is either already loaded, or available. If a module can be found, and it not yet loaded, it is automatically loaded.

The return value is a boolean status code. No error is generated when the database type cannot be resolved to a module.

Example:

```
dbx defined mysql
```

dbx exists

```
dbx exists dbtype
d.exists()
Dbx.Exists(dbtype)
```

Check whether a driver module for the specified database type is already loaded. No attempt is made to auto-load a module if it is not already loaded.

The return value is a boolean status code. No error is generated when the database type cannot be resolved to a module.

Example:

```
dbx exists postgresql
```

dbx get

```
dbx get dbtype attribute
d.get(attribute)
d.attribute
d[attribute]
Dbx.Get(dbtype,attribute)
```

Get an attribute the database driver module. The following attributes can be queried:

- *aliases*
  A list of recognized alias names of the database type.
- *builtin*
  A flag indicating whether this driver module is built-in. Built-in module cannot be unloaded.
- *address_city*
  The city part of the author contact address.
- *address_country*
  The country part of the author contact address, following the ISO3166 standard.
- *address_state*
  The state part of the author contact address. Empty if not applicable.

- *address_street*
  The street address part of the author contact address. Includes floor, house number, etc.
- *address_zip*
  The **ZIP** code or other applicable postal code of the author contact address.
- *affiliation*
  The institution the author works for.
- *affiliationduns*
  The **DUNS** registration ID of the affiliated institution. This is primarily useful for US government projects.
- *affiliationurl*
  The **URL** of the affiliated institution.
- *author*
  The author of the module, as free-format text.
- *authorurl*
  A **URL** with information on the author, or an empty string if unset.
- *blobsize*
  The default maximum binary blob size.
- *category*
  A category string to be used if the module is stored in a repository.
- *classuuid*
  The base class **UUID** of this database interface module
- *comment*
  A free-form comment
- *date*
  The date the module source code was last changed.
- *doi*
  A digital object identifier for the module, if defined.
- *email*
  An email contact address of the developer of the module.
- *flags*
  A collection of flags indicating special capabilities of the module. The only flag currently used is *sqlarrays*, indicating support for arrays as column data types.
- *history*
  Module history data
- *infourl*
  A **URL** with information on the module, or an empty string if unset.
- *keywords*
  A list of keywords associated with the module.
- *license*
  The license class associated with this module. Setting the license to a standard type updates the associated **URL** with a standard location.

- *licenseurl*
  A **URL** with details about the module license.
- *literature*
  A free-form literature reference.
- *name*
  The official name of the module. Since the information may be queried via an alias name, this can be different from the command argument.
- *objectfile*
  The name of the currently loaded object or shared library file implementing the functions of the selected database interface module.
- *orcid*
  The **ORCID** code of the author (see www.orcid.org).
- *path*
  The repository path for displaying hierarchical repository trees. This attribute is independent of any file system paths.
- *phone*
  A contact phone number of the author.
- *port*
  The default **TCP/IP** port used for communication with database servers.
- *protocolversion*
  The primary protocol version this module supports.
- *references*
  Cross references of the module. This is a nested list of class **UUID**s and reference type tags.
- *regid*
  For officially registered data driver modules handlers, this is the assigned registration ID. Unregistered modules report zero.
- *slot*
  The driver table slot this module is loaded into.
- *sourcefile*
  The name of the source file for the database interface module.
- *textsize*
  The default maximum text blob size.
- *version*
  Version information for the module.This is a string in a 1.2.3 (or shortened) style.
- *versionuuid*
  The **UUID** associated with this module version.

Example:

```
dbx get mysql version
```

dbx list

```
dbx list ?pattern?
Dbx.List(?pattern=?)
```

Return a list of all currently loaded database driver modules. The output may be filtered by a string pattern. Its syntax is the same as in standard TCL command **`string match`**.

dbx load

```
dbx load dbtype ?objectfile?...
dbx load all
Dbx.Load(dbtype,?objectfile?,...)
Dbx.Load(“all”)
```

Load or re-load a database interface module. If no object file name is specified, the name of the shared library, DLL or bundle is automatically constructed from the database type name, and the module is located by traversing the database module path, which is accessible via the control array element *::cactvs(databasepath)*, but this mechanism can be overridden by specifying an explicit object file with or without a path.

The *all* command variant locates all currently available database interface modules in the module search path and loads these. This is primarily useful for automatic database interface attribute documentation.

Example:

```
dbx load mysql
```

Above statement locates and loads the standard driver for interacting with **MYSQL** and **MARIADB** databases. Depending on the platform, the object file would be named *dbx_mysql.so*, *dbx_mysql.dll,* etc.. It is located in the module directory of a standard Cactvs distribution.

In case the interface module is already loaded, the current module is unloaded first, so this command can be used to update a driver in a running application. Nevertheless, swapping a driver while database access objects which rely on this driver are in existence in the current process is usually not a good idea, though the details on whether this is possible or not depend on the module implementation.

For TCL, the return value of the command is the slot in the handler module table the module has been loaded into. This corresponds to the value of the *slot* attribute which can be queried via **`dbx get`**. For PYTHON, the return value is a module reference.

dbx ref

```
Dbx.Ref(dbtype)
```

PYTHON-only method to get a reference of the module, which allows terser attribute retrieval commands and other operations.

dbx subcommands

```
dbx subcommands
dir(Dbx)
```

This command returns a list of all the defined subcommands of the **`dbx`** command.

The filex Command

The filex command manages chemical structure and reaction file I/O modules. In many cases, actively loading of I/O modules is not required because of the built-in auto-load mechanism. If the toolkit encounters a file of unknown type, an attempt is made to load a suitable module by constructing the name of the module from the file suffix. However, that mechanism fails in case the file does not have a suffix, or a non-standard suffix, or the data source is not a file but some other stream, such as a network connection, a pipe, or a standard I/O channel. In these cases, explicit managing of I/O modules is required.

The **`filex`** command has the following subcommands:

filex defined

```
filex defined format
Filex.Defined(format)
```

A check to determine whether the specified format is supported by an I/O module. In case the appropriate handler is not yet loaded, an attempt at auto-loading is made. For the equivalent command without auto-loading, see **`filex exists`**. The result value is a boolean status code.

filex exists

```
filex exists format
Filex.Exists(format)
```

A check to determine whether an I/O module for the specified format is currently loaded. This command variant does not attempt auto-loading. The format name may be either the primary name of a loaded module, or any of alias format name aliases the module recognizes. For the equivalent command with auto-loading, see **`filex defined`**. The result value is a boolean status code.

filex get

```
filex get format attribute
fx.get(attribute)
fx.attribute
fx[attribute]
Filex.Get(format,attribute)
```

Query the value of an attribute of the I/O module. The list of attributes is detailed in the paragraph on the **`filex set`** command.

In case the format argument cannot be resolved by an active module, an attempt to auto-load a suitable module is made.

filex list

```
filex list ?pattern?
Filex.List(?pattern=?)
```

List the names of all currently loaded I/O modules. A string match pattern may be used to filter the result list. The variant **`filex modules`** is an alias to this command.

filex load

```
filex load format ?objectfile?
filex load all
Filex.Load(format,?objectfile?,...)
Filex.Load("all")
```

Explicitly load an I/O module. If the module is already loaded, the current version is unloaded first. If no specific object file (a shared library on Unix/Linux, a DLL on Windows, a bundle file for OSX) is specified, the standard name of the module file is automatically constructed from the format name, and then the file searched in the directories in the I/O module path. The module path can be customized in the control variable **`::cactvs(filexpath)`**.

For **Tcl**, the return value of the command is the slot in the module table the module has been loaded into. This corresponds to the value of the *slot* attribute which can be queried via **`filex get`**. For **Python**, the return value is a module reference.

The second form of the command scans the currently set I/O module extension search path and loads all accessible modules which are not yet in memory. Modules which are already active in the running application are not unloaded, and only a single instance of each I/O module, even if present under various alias names in the module directories, is loaded. This form of the command does not return a value.

filex modules

```
filex modules ?pattern?
Filex.Modules(?pattern=?)
```

This is an alias for **`filex list`**.

filex ref

```
Filex.Ref(format)
```

Python-only method to get a reference of the module, which allows terser attribute retrieval commands and other operations.

filex reload

```
filex reload format ?objectfile?
Filex.Reload(format,?objectfile?,...)
```

A variant of the **`filex load`** command which fails if the I/O module was not previously loaded. There is no *all* variant of this command.

filex set

```
filex set format ?attribute value?...
filex set format dict
fx.set(?attribute,value?,...)
fx.set(dict)
fx.attribute = value
fx[attribute] = value
Filex.Set(format,?attribute,value?,...)
Filex.Set(format,dict)
```

Set attributes of the I/O module. Compared to other classes of modules, there are rather few attributes in a module which can be set in a meaningful manner. Some of the listed attributes are read-only. They are included in this section because it is cross-referenced from the **`filex get`** command. These are the supported attributes:

- *address_city*
  The city part of the author contact address.
- *address_country*
  The country part of the author contact address, following the ISO3166 standard.
- *address_state*
  The state part of the author contact address. Empty if not applicable.
- *address_street*
  The street address part of the author contact address. Includes floor, house number, etc.
- *address_zip*
  The **ZIP** code or other applicable postal code of the author contact address.
- *affiliation*
  The institution the author of the module works for.
- *affiliationduns*
  The **DUNS** registration ID of the affiliated institution. This is primarily useful for US government projects.
- *affiliationurl*
  The **URL** of the affiliated institution.
- *aliases*
  A list of alternative names of for the formats the module supports.
- *author*
  The author of the module.
- *authorization*
  An authorization string, for example a service login **URL**. This is for example used in the *dropbox* meta I/O module. In that case, it is a Web **URL** generated by the module from the compiled-in application secret. Using that **URL**, the user must log into a **DROPBOX** account and approve access to the files of that account by the application. Only after this has been performed, opening **DROPBOX** files with a **`molfile open`** command succeeds.
- *authorurl*
  A **URL** with information on the author, or an empty string if unset.
- *builtin*
  A boolean read-only boolean flag indicating whether the module is built-in.
- *capabilities*
  A list of features and behaviors the I/O module supports. Only a few of the flags which can be found here can be changed in a productive fashion. These include:

  disabled
  Temporarily disable this module, without unloading it

  nommap
  Never attempt to memory-map files of this format

- *category*
  A category string to be used if the module is stored in a repository.
- *classuuid*
  The base class **UUID** of this module.
- *comment*
  A free-form string comment on the module.
- *date*
  The data the module source was last modified.
- *doi*
  A digital object identifier for the module, if defined.
- *email*
  The email address of the author of an I/O module.
- *ensproperty*
  The name of a property which is used to store structure information in the file. This is only used for file formats where storing structure data is a minor objective, not for standard chemical structure exchange formats.
- *functions*
  This attribute is a read-only list of the classes of available functions in the function table of the module. Developers can use this information to determine whether a module is input-only or output-only, or supports acceleration methods for scanning structure files.
- *id*
  The internal format ID of the module in the current program run. This is usually identical to the slot in the extension table for module was loaded or compiled into.
- *infourl*
  A **URL** with information on the module, or an empty string if unset.
- *keywords*
  A list of keywords associated with the module.
- *license*
  The license class associated with this module. Setting the license to a standard type updates the associated **URL** with a standard location.
- *licenseurl*
  A **URL** with details about the module license.
- *literature*
  A free-form literature reference.
- *mimetype*
  The **MIME** type associated with the file format, for example *chemical/x-mdl-molfile*. This information is used for constructing **HTTP** headers for data transfer in Web environments and similar tasks.

- *name*
  The primary name of the format the I/O module handles.

- *nitrostyle*
  The style of nitro groups and similar groups in the file, i.e. whether these are preferably encoded with pentavalent nitrogen or a charge pair. Possible values are *asis* (does not matter, or unknown), *ionic*, *neutral*, *xionic* and *xneutral*. If this value is not *asis*, structures written to the file are automatically adapted. This is performed on duplicates of the output structures, so the objects used in a **`molfile write`** or similar command does not change. On the other hand, the requirement to duplicate the object, manipulate the duplicate, and destroy it after it has been used can be time-consuming.

- *objectfile*
  The full path name of the loaded object file or dynamic library. This attribute is read-only.

- *orcid*
  The **ORCID** code of the author (see www.orcid.org).

- *parameters*
  A dictionary of format-specific keyword/value pairs which are not represented as a general **`molfile`** object attribute. When a file of a specific format is opened, the data from the corresponding I/O module is copied to the *parameters* attribute of the **`molfile`** object, where it may be further customized by **`molfile set`** commands before an input or output operation. Changing this attribute in the I/O module modifies the initial content of the parameters attribute of all **`molfile`** objects associated with this format created in the future. Explicitly changing the format of a **`molfile`** object refreshes the parameter set.
- *path*
  The repository path for displaying hierarchical repository trees. This attribute is independent of any file system paths.
- *phone*
  A contact phone number of the author.

- *reactionproperty*
  The name of a property which is used to store reaction information in the file. This is only used for file formats where storing reaction data is a minor objective, not for standard chemical structure exchange formats.

- *readflags*
  A list of flags to adjust input behavior. Not all flags are supported for all I/O modules. Unsupported flags are silently ignored. The flag set is copied as default to any **`molfile`** object which uses the I/O handler module. The flag set is the same as for the **`molfile`** *readflags* attribute, but only a subset of these flags make sense as presets. The flags can be modified on the I/O module level if desired:

  - *none*
    The same as an empty list; no flags are set.

- *aroresolver*
  If set, resolve bonds marked in the file as *aromatic* into a Kekulé system. This includes resolution of bonds which are explicitly marked as query bonds (i.e. bond type 4 in **MDL** *Molfiles*). This is very useful to fix frequently seen MDL Molfiles which encode structures, not queries, but nevertheless use an aromatic bond type in violation of the file format specification. Aromatic system resolution works much more robustly for structures with a complete set of hydrogens. It is advisable to combine this flag with automatic hydrogen addition.

- *autowrap*
  If set, the file is automatically rewound if the end of the file is reached, and the start record of the operation has not yet been encountered again. This behavior only applies to the **`molfile scan`** command, not to normal record input. Wrapping is not possible on data source which cannot be rewound.

- *basiconly*
  Only read basic connectivity information, but not additional properties. Supported only on formats which use the native **CACTVS** structure data storage system (*cbin, cbs, bdb*).

- *chargebalancer*
  If set, perform a charge balancing step after reading, in an attempt to obtain a neutral structure.

- *chargecombiner*
  If set, perform a charge combination step after reading, in an attempt to obtain a neutral structure.

- *complexresolver*
  If set, try to resolve a purely VB-based structure representation into a representation which utilizes *complex* bonds for bonds between ligands and metal centers which cannot be described well with electron-counted VB bonds.

- *fixdoublespace*
  If set, this flag instructs I/O modules with support for this feature to read structure files which contain one spurious empty line after each data line, which unfortunately appears to happen sometimes when **DOS**-encoded files are transferred to Apple systems. This is not the same as reading **CR/LF** files on **CR**-only or **NL**-only platforms, or vice versa, which is always possible and fully automatic. This flag addresses the problem that, due to mishandling by obscure transfer software, duplicated **EOL**-markers are introduced in the file (two identical **CR/LF**, or **CR**, or **NL** pairs after each data line).

- *fixstereo*
  If set, remove spurious atom and bond stereo descriptors assigned to non-stereogenic centers.

- *fixwedges*
  If set, invert wedge bonds encoded with the base at the stereo center to the IUPAC-conforming style with the tips at the stereo center.

- *ignoreempty*
  If set, records which do not contain any atoms are silently ignored and the next record with atoms is returned instead.

- *ignoreerrors*
  If set, ignore records with raise errors on file input. Instead, silently attempt to re-synchronize the read pointer and proceed with the next record, until the end of the file has been reached, or an undamaged record could be read successfully.
- *ignorevisibility*
  Ignore any object visibility information in the file and read all data as visible objects.
- *ignorecr*
  Allow an isolated carriage return (**ASCII** 13) character without following `NL` (**ASCII** 10) character as data content instead of examining it as potential line break symbol. This flag is necessarily ignored on Mac-style input files which only use `CR` as `EOL` markers.
- *ignoreeitherdb*
  If set, ignore the *either* flag for double bonds when reading **MDL Molfiles**. The default is to translate it into the *crossed* bit of the `B_FLAGS` property.
- *keepcoords*
  If set, always keep atomic 2D layout coordinates, even if they are, for example in reactions, overlapping on the reagent and product sides. By default the coordinates of molecules are adjusted if necessary to be non-overlapping. This is done by moving molecules only, not by scaling the coordinates, and never by recomputing any coordinates.
- *latehprocessing*
  If set, hydrogen modification (addition, deletion) is performed after standardization operations (see various *resolver* attributes). By default, hydrogen addition is performed before these routines are called.
- *mergedata*
  If this flag is set, multi-line input from SD file data lines into a simple string property is merged into a single string value, with tab characters indicating the newlines in the file. By default, in such cases every line of a multi-line data item is stored as a new property instance. This is equivalent to the property attribute *mergedata* (see **`prop set`** command).
- *multibondcheck*
  If set, attempt to intelligently resolve any atoms with excessive multiple bonds consuming bond electrons in excess of the available number by recoding such bonds as charge pairs.
- *nocoordinatecheck*
  If set, no attempt is made to add missing coordinates, for example for automatically added hydrogen atoms, to the 2D and 3D coordinate sets, if such coordinates were present in the original record.
- *noimplicith*
  Do not add implicit hydrogen. This flag only applies to file formats which exactly define a default number of hydrogens (for example, **SMILES**) as implicit part of the structure . It has no effect in file formats which just tend to omit hydrogen (for example, **MDL** *Molfiles*).

- *nometal*
  Assert to the file input routine that the input does not contain any metal atoms. In that case ambiguous atom symbols, for example *CA* or *CD in* **PDB** *files,* are interpreted as carbon (in alpha and delta position), and not as calcium or cadmium.

- *nometalh*
  Assert that none of the metal atoms in the structure has any missing hydrogen ligands. If set, hydrogen addition, if selected, skips the processing these atoms.

- *noorigin*
  By default, every ensemble or reaction read from a file is augmented with a property `E_FILE` or `X_FILE`, indicating the origin of the record by recording the file name, record number and other information in the automatically attached property. If this information is not of interest, this wasteful step can be suppressed by setting the flag.

- *noradicals*
  Assert that the file does not contain any radicals. This can for example be helpful in the resolution of aromatic systems (see *aroresolver* attribute).

- *pedantic*
  Strictly adhere to the format specification and flag any deviation as error. This is feature is only well implemented for **MDL MOLFILES**. It is intended to be used for strict format checking.

- *radicalcharger*
  Edit radicals which are typically formed by reading a file without formal atomic charge information by adding standard formal charges, for example replacing NR_4 with $N^{(+)}R_4$ and OR with $O^{(-)}R$. This only works reasonably well if the file contains a complete hydrogen set.

- *readas2d*
  Force interpretation of the atomic coordinates in the record as 2D display coordinates (property `A_XY`), even if syntax or data items in the file indicate the presence of 3D coordinates. This is useful for simple reading of records where 3D coordinate fields were abused for storing display coordinates.

- *readparity*
  For **MDL MOLFILES**, read the parity information. By default, as recommended by **MDL**, this information is not read and parity is instead computed from wedges if needed.

- *simpleradicals*
  Assume that any radical encountered is a singlet, and not anything more complex such as triplets etc., regardless what the file encodes.

    - *tautoresolver*
      Perform a tautomer standardization on the read structure. This operation invalidates numerous atom and bond properties, such as coordinates, but in this special case all ensemble properties which were attached to the processed structure are retained, regardless of their sensitivity toward atom and bond changes. Tautomer resolution requires a complete hydrogen set, so either these must be present in the input file, or a suitable hydrogen addition mode must have been set on the file handle. The processing behind this input option is comparatively expensive. For normal input, when speedy input and maximum fidelity of the data to the original file is desired, this flag should not be set.

- *regid*
  A numerical registration ID assigned to registered modules.

- *references*
  Cross references of the module. This is a nested list of class **UUID**s and reference type tags.

- *slot*
  The slot the module was loaded into. This attribute is read-only.

- *sourcefile*
  The name of the source file of the module. This attribute is read-only.

- *suffixes*
  A list of the file suffixes this module recognizes as typical for the implemented format. If a file with a suffix is opened for writing without specifying an explicit format, the last loaded module which has the suffix in its list determines the automatically assigned format. Suffixes are ignored as format identifier for file input and updates. In these cases, the file contents are analyzed to determine the format. This attribute is read-only.

- *version*
  The version of the module. This is a string in a 1.2.3 (or shortened) style.

- *versionuuid*
  The version **UUID** associated with this module version.

In case the format argument cannot be resolved by an active module, an attempt to auto-load a suitable module is made.

filex subcommands

```
filex subcommands
dir(Filex)
```

List all supported subcommands of the **`filex`** command in an installation.

filex unload

```
filex unload ?format?...
fx.unload()
Filex.Unload(?format?,...)
```

Unload zero or more I/O modules. It is an error to specify the name of a module which is not loaded.

Built-in I/O modules cannot be unloaded. If the use of one of these needs to be switched off, it is possible to set the *disabled* flag of the *capabilities* module attribute via the **`filex set`** command.

The command returns the number of unloaded I/O modules.

The filter Command

The **`filter`** command is used to manage filter objects. Filter objects are a convenient method to quickly access subsets of objects, for example subsets of atoms or bonds with specific properties.

A filter basically consists of the name of a property which is used as foundation for the comparison. It also defines a comparison operator and one or more comparison values. Objects whose property values of the filter property pass the test are passed on to further processing.

The **`filter`** command has the following subcommands:

filter create

```
filter create filtername ?attribute value?...
filter create filtername dict
Filter(filtername,?attribute,value?...)
Filter(filtername,dict)
Filter.Create(filtername,?attribute,value?...)
Filter.Create(filtername,dict)
```

Create a new filter. In case the filter already exists, this is the same as **`filter set`**. A default filter without any other configured attributes does nothing and lets everything pass. The supported attributes and values are explained in the paragraph on the **`filter set`** command.

The return value is the filter name or reference.

filter defined

```
filter defined filtername
Filter.Defined(filtername/fref)
```

A boolean check whether the filter is available. In case it is not yet in memory, an attempt is made to auto-load or auto-instantiate it. For a command variant without auto-loading, see **`filter exists`**.

filter exists

```
filter exists filtername
f.exists()
Filter.Exists(filtername/fref)
```

A boolean test whether the filter is current defined and loaded. No attempt is made to auto-load it. For a command with auto-loading, see **`filter defined`**.

filter delete

```
filter delete ?filtername?..
f.delete()
Filter.Delete(?filtername/fref?,...)
```

Delete zero or more filters. Note that it *is* possible to delete built-in filters. An attempt to delete a non-exiting filter raises an error. The return value of this command is the number of deleted filters.

filter get

```
filter get filtername attribute
f.attribute
```

```
f[attribute]
f.get(attribute)
Filter.Get(filtername/fref,attribute)
```

Query an attribute from a filter definition. The supported attributes are detailed in the paragraph on the **`filter set`** subcommand.

If the specified filter is not yet loaded, an attempt to auto-load a definition file is made.

filter list

```
filter list ?pattern?
Filter.List(?pattern=?)
```

Get a list of all currently loaded filters, including the built-in filter definitions. If desired, a string match pattern can be supplied to filter the reported set.

filter load

```
filter load filtername
filter load all
Filter.Load(filtername)
Filter.Load("all")
```

Attempt to explicitly load a filter via the auto-loader mechanism. If the filter is already in memory, the command has no effect. In case auto-loading fails, an error results. The return value is the filter table slot index the filter is loaded into for **Tcl**, or a filter reference for **Python**.

The second form of the command scans the currently set filter search path and loads all accessible filters which are not yet in memory. Filters which are already active in the running application are not unloaded, and only a single instance of each I/O filter, even if present under various alias names in the filter directories, is loaded. This form of the command does not return a value.

filter query

```
filter query keyword ?objectclass? ?mode? ?casesensitivity?
Filter.Query(keyword=,?objclass=?,?mode=?,?casesensitivity=?)
```

Search the internal filter database by matching the keyword against a standard set of filter attributes, such as name, description, keywords, category, comment and **UUID**s. Only the current memory database is checked, no auto-loading or repository checks are performed.

By default all filter definitions are matched. The object class argument (such as *atom*) can be used to limit the search to filters using a property of a specific property class. Providing an empty argument is the same as omitting the argument.

The optional mode argument changes the string comparison mode. The default is *equal*, other possibilities are *substring*, *left* (match beginning of string), *right* (match end of string), *like* (as the **SQL** operator), *glob* or *regexp*.

The final argument can be *case* (case-sensitive matching) or *nocase* (case-insensitive comparison, this is the default).

The return value is a list of the names or references of the matched filters.

filter read

```
filter read filename/dirname
filter read all
Filter.Read(filename=)
Filter.Read("all")
```

Read a filter definition file with one or more filter definitions, a directory containing such definition files, or the filter definition files contained in all accessible directories found in the filter auto-load path. In case a filter is already defined in the current interpreter, its definition is overwritten by what is found in the input file.

The return value is a list of two elements. The first element is the total number of filter definitions read from the file. The second element is the name (**Tcl**) or reference (**Python**) of the first filter read.

filter readblob

```
filter readblob datablob
Filter.Readblob(data=)
```

Read a filter definition from an in-memory **XML** blob. This blob can only contain a single filter definition. In the **Python** case, the data blob can be either a **Unicode** string, or a byte array.

The return value is the name (**Tcl**) or reference (**Python**) of the newly read filter definition.

filter ref

```
Filter.Ref(filtername)
```

Python-only method to get a reference of the filter, which allows terser attribute retrieval commands and other operations.

filter set

```
filter set filtername ?attribute value?...
filter set filtername dict
f.set(?attribute,value?,...)
f.set(dict)
f.attribute = value
f[attribute] = value
Filter.Set(filtername/fref,?attribute,value?,...)
Filter.Set(filtername/fref,dict)
```

Set one or more filter attributes. Some of the attributes listed below are read-only. They are included here because the **`filter get`** subcommand refers to this section. The following attributes are supported:

- *address_city*
  The city part of the author contact address.
- *address_country*
  The country part of the author contact address, following the ISO3166 standard.
- *address_state*
  The state part of the author contact address. Empty if not applicable.
- *address_street*
  The street address part of the author contact address. Includes floor, house number, etc.

- *address_zip*
  The **ZIP** code or other applicable postal code of the author contact address.
- *affiliation*
  The institution the author of the filter definition works for.
- *affiliationduns*
  The **DUNS** registration ID of the affiliated institution. This is primarily useful for US government projects.
- *affiliationurl*
  The **URL** of the affiliated institution.
- *aliases*
  A list of alias names for the filter.
- *author*
  The name of the author of the filter definition as free text.
- *authorurl*
  A **URL** with information on the author, or an empty string if unset.
- *category*
  A category string to be used if the filter is stored in a repository.
- *classuuid*
  The base class **UUID** of this filter.
- *comment*
  A free-form comment string.
- *date*
  The date of the last change in the filter definition.
- *description*
  A short single-line description of the filter, suitable for use in menus and similar circumstances.
- *doi*
  A digital object identifier for the filter, if defined.
- *email*
  An email contact address of the author of the filter.
- *field*
  The name of a field of the filter property. In case the complete property is used, which is the default, this is an empty string. This field may also be set directly by using field notation in the *property* attribute.
- *fieldindex*
  In case a only a field of the filter property is used for comparison, this is the numerical index, starting with 0, of the field in the property. This is a read-only attribute and automatically updated then setting the field attribute. If the complete property data is used, by the filter, the value of this attribute is -1.

- *file*
  The full path name of the file the filter definition was read from, or an empty string if the filter is built-in or defined in the script. This is a read-only attribute.
- *flags*
  Various flags to modify the operation of the filter. The flag set can be any combination of words from the set
  - *notavail_fail*
    If set, the filter fails if the filter property cannot be computed, but no error is raised
  - *notavail_pass*
    If set, filter always passes if the filter property cannot be computed, but no error is raised
  - *nocompute*
    If set, do not attempt to compute the filter property on the filtered object, if it is not yet present, this condition is treated as a failed computation
  - *recompute*
    If set, the filter property is recomputed once when the filter data is prepared for a filter operation. For example, the property of an atom filter is re-computed once before an `ens atoms` command, but not for the filter test of each individual atom.

  The *get* variant of this command can additionally return several flag words describing the current internal state of the filter definition.
- *infourl*
  A **URL** with information on the filter, or an empty string if unset.
- *keywords*
  A list of keywords associated with the filter.
- *license*
  The license class associated with this filter Setting the license to a standard type updates the associated **URL** with a standard location.
- *licenseurl*
  A **URL** with details about the filter license.
- *literature*
  A free-form literature reference.
- *mode*
  The filter operation mode. This is an important, but complex attribute. The default value is *simple*, and this is what is needed in almost all standard applications. The possible values for this attribute are:
  - *simple*
    Straightforward application of the filter. The property values of the filtered chemistry objects are directly compared to the filter values. In case the filter property has a width of more than one slot (see *width* attribute of property definitions), it is sufficient if any of the multiple values passes the filter.

  - *all*
    Only usable for properties with a width of more than one. In this mode, all of the multiple property slot values of a chemistry object must pass the filter.
  - *diff*
    Only usable for properties with a width of more than one. In this mode, among the multiple property slot values on a tested object there must be some which pass, and some which fail.
  - *allatoms*
    Instead of using the filtered chemistry object, use all atoms the chemistry object contains. All atoms must pass the filter condition.
  - *someatoms*
    Instead of using the filtered chemistry object, use all atoms the chemistry object contains. Some atoms must pass the filter condition.
  - *noatoms*
    Instead of using the filtered chemistry object, use all atoms the chemistry object contains. No atoms must pass the filter condition.
  - *diffatoms*
    Instead of using the filtered chemistry object, use all atoms the chemistry object contains. There must be atoms which pass the filter as well as atoms which fail.
  - *allneighbors*
    Instead of using the filtered chemistry object, use all neighbor objects of the same type. All neighbors must pass the filter condition. Currently, neighborship is only defined for atom and bond objects.
  - *someneighbors*
    Instead of using the filtered chemistry object, use all neighbor objects of the same type. All neighbors must pass the filter condition. Currently, neighborship is only defined for atom and bond objects.
  - *noneighbors*
    Instead of using the filtered chemistry object, use all neighbor objects of the same type. All neighbors must pass the filter condition. Currently, neighborship is only defined for atom and bond objects.
  - *diffneighbors*
    Instead of using the filtered chemistry object, use all neighbor objects of the same type. There must be neighbors that pass the filter condition as well as neighbors which fail. Currently, neighborship is only defined for atom and bond objects.
- *name*
  The primary name of the filter. Since alias names are also used to resolve a filter reference, this may be different from the argument supplied in the command.
- *operator*
  The operator used for comparing the filter value(s) to the property values of the compared chemical objects. It can be one of

    - *exact*
      object and primary filter value must match
    - *smaller*
      object value must be smaller than the primary filter value
    - *larger*
      object value must be larger than the primary filter value
    - *range*
      object value must be between the primary and secondary filter values
    - *not*
      object and primary filter value must not match
    - *bitset*
      object value must have all on bits in the primary filter value also set
    - *bitunset*
      object value must not have any bit of the on bits in the filter value set
    - *alternative*
      object value must be equal to the primary or secondary filter value
    - *neither*
      object value must be different both from both filter values
    - *le*
      object value must be larger or equal to the primary filter value
    - *ne*
      object value must be smaller or equal to the primary filter value

    The standard mathematical operator notation >, >=,== etc. may also be used to identify the operator.

- *orcid*
  The **ORCID** code of the author (see www.orcid.org).
- *path*
  The repository path for displaying hierarchical repository trees. This attribute is independent of any file system paths.
- *phone*
  A contact phone number of the author.
- *property*
  The name of the property associated with the filter. In the context of the **`filter create`** or **`filter set`** commands, the property name argument may include a field component. If a field is specified as part of the name, the *propertyfield* and *propertyfieldindex* attributes are also set as a side effect. Example:

```
filter create alphac property A_RESIDUE(atomtag) value “CA” operator =
```

This defines a filter *alphac* which selects all atoms which have the value *CA* in the field *atomtag* of the A_RESIDUE property, which is for example filled when reading **PDB** files. With a **filter get** command, the reported property name is still only A_RESIDUE. The field information can be obtained via the *propertyfield* attribute.

- *propertyfield*
  This is a read-only attribute. It reports the name of the field of the filter property, if such a field was specified. For filters which do not use fields, the result is an empty string.
- *propertyfieldindex*
  This is a read-only attribute. It returns the index of the field of the filter property, if such a field was specified. For filters which do not use fields, the result is minus one.
- *references*
  Cross references of the filter. This is a nested list of class **UUID**s and reference type tags.
- *regid*
  If the filter is registered, this is its registration ID. Unregistered filters have a zero registration ID.
- *slot*
  This is a read-only attribute. It reports the index of the filter definition in the filter table. This is useful for debugging only.
- *value1*
  The primary filter comparison value. This may also be simply written as *value* without the rank indicator. The value can be specified in any notation which can parsed as data value of the filter property. Changed filter values are parsed after the **filter set** or **filter get** command has processed all its arguments. The order of arguments does not matter, but at the end of the command, the property and data value must be compatible.
- *value2*
  The secondary filter comparison value. The same restrictions apply as for the primary filter comparison value.
- *value3*
  The auxiliary filter comparison value. The same restrictions apply as for the primary filter comparison value. This value is not used directly in conjunction with any of the scriptable comparison operators, but it does have a role in some internal functions.
- *values*
  Return a list of the filter comparison values (individually accessible as *value1..3* attributes) which are actually used in this filter definition. This is a read-only attribute.
- *version*
  The version of the filter definition. This is a string in a 1.2.3 (or shortened) style.
- *versionuuid*
  The version **UUID** associated with this filter version.

If the filter argument is not yet loaded, an attempt to auto-load the definition file is made.

filter string

```
filter string filtername
f.string()
Filter.String(filter=)
```

Encode the filter as **XML** blob and return the blob as result. The blob format is the same as output by the **`filter write`** command.

filter subcommands

```
filter subcommands
dir(Filter)
```

Get a list of all supported subcommands of the **`filter`** command in this particular application.

filter write

```
filter write filtername ?filename?
f.write(?filename=?)
Filter.Write(filter=,?filename=?)
```

Write the current definition of the filter to a file in **XML** format. The file can be loaded into future script interpreters explicitly (see **`filter read`** command) or implicitly via the filter definition auto-load mechanism. If no file name is given, the name of the file is automatically constructed from the filter name in lower case and the suffix *.fil.* In addition to normal file names, the magic names *stdout* and *stderr* may be used, as well as already opened **Tcl** or **Python** socket and file handles, plus pipes indicated by a file name which starts with “|”. Writing to **Tcl** channels is not supported on the MS Windows platform.

The command returns the (possibly auto-generated) file name.

It is possible and sometimes useful to write out built-in filter definitions.

The json Command

The **`json`** command is used to facilitate network communication by means of exchanging **JSON-RPC** messages. **JSON** message objects are also useful to parse other styles of **JSON**-formatted data which are not really **JSON-RPC** but use basic **JSON** encodings.

JSON objects are created like other toolkit objects, and have internal state, so it is possible to communicate with multiple sources simultaneously by creating a **JSON** object for every channel.

JSON objects share many characteristics with **SOAP** objects, and many commands are identical or at least very similar.

Only interpreters compiled with **JSON** support contain this command.

These are the currently supported **JSON** object commands:

json append

```
json append jsonhandle ?attribute value?...
json append jsonhandle dict
j.append(?attribute,value?,...)
j.append(?attribute=value?,...)
j.append(dict)
```

This is a variant of the **`json set`** command. The difference is that the supplied data is appended to the current attribute value instead of replacing it. In case appending is not a possible operation, the result is the same as using **`json set`**.

The set of supported attributes is explained in the paragraph on **`json set`**.

The command returns the original handle or reference.

json create

```
json create ?attribute value?...
json create dict
Json(?attribute,value?,...)
Json(dict)
Json(?attribute=value?,...)
Json.Create(?attribute,value?,...)
Json.Create(dict)
Json.Create(?attribute=value?,...)
```

Create a new **JSON** object. The return value is the object handle or reference. If no additional attributes are specified, an object with default settings is created. Processing of specified optional attribute/value pairs is performed in as an identical fashion to the **`json set`** command.

json delete

```
json delete ?jsonhandle?...
json delete all
j.delete()
Json.Delete(?jsonhandle/jref?,...)
Json.Delete("all")
```

Destroy one or more **JSON** objects. The special handle *all* can be used to remove all **JSON** objects currently existing in the application.

For the sake of consistency with commands of similar objects, **`json close`** is an alias to this command.

The return value is the number of successfully deleted **JSON** objects.

json error

```
json error jsonhandle ?errormessage? ?channel? ?errorcode? ?errordata? ?id?
j.error(?errormessage=?,?channel=?,?errorcode=?,?errordata=?,?id=?)
```

Assemble and potentially send a properly formatted **JSON-RPC** error message. The formatted message is stored in the *result* attribute of the **JSON** object, returned as command result, and also sent via the channel if either the argument is a valid **Tcl** or **Python** channel handle, or the object has an associated channel in the internal attribute set.

All message parameters are taken from the internal object attribute set if they are not explicitly set in the arguments. If arguments are set, the corresponding object attributes are also updated. Any arguments must be properly formatted - the error code and ID are an integer, the error message a simple string, and the error data a complete **JSON** object encoding.

Since this command is a reply to a specific message ID, the internal object ID attribute is not incremented, as in the **`json request`** command.

json get

```
json get jsonhandle attribute
j.get(attribute)
j.attribute
j[attribute]
```

Query the value of an attribute of a **JSON** object. The list of recognized attributes is explained in the paragraph on the **`json set`** command.

The return value of the command is the value of the attribute.

json list

```
json list ?pattern?
Json.List(?pattern=?)
```

Return a list of the handles of all currently existing JSON objects in the application. If desired, the list can be filtered by a string pattern.

json parse

```
json parse jsonhandle ?data?
j.parse(?data=?)
```

Parse a **JSON-RPC** message. If the data argument is set, its value is used. Otherwise the parsed data is the value currently stored in the *body* object attribute, which is for example set by the **`json read`** command. The command does not replace the body attribute of the object if an explicit argument is set.

The command resets the ID, error code, error message, error data and result attributes of the object and then re-populates those object fields for which data is found in the message.

The return value of the command is one, if the message was a valid result reply, and zero if the message was an error report. In case of a syntax error in the parsed data, a **Tcl** or **Python** error is generated.

After parsing, the *result* and *errordata* object attributes, if they were set during the parse, are still **JSON** object encodings. In most cases these are further dissected by the application of a **`decode -json`** command.

json read

```
json read jsonhandle ?channel?
j.read(?channel=?)
```

Read one **JSON-RPC** message from the channel. If no channel argument is supplied, the internal channel handle of the object is used. It the channel argument is set, the internal attribute is also updated.

The command recognizes **HTTP** headers in the input stream and stores these separately from the message body in the internal *header* object attribute. Only a single message is read from the channel, so in case there are multiple messages queued, the command must be repeatedly invoked. The complete message text, without header if there was one, is stored in the *body* object attribute.

The command returns the full message text, which is the same as the *body* attribute. It is not parsed or analyzed further. This command does not alter the *errorcode, errormessage, errordata* or *result* fields of the object. In most cases, the next step after reading a message is to analyses it with a **`json parse`** command.

json ref

```
Json.Ref(identifier)
```

Python-only method to get a reference of the **JSON** object from its handle.

json request

```
json request jsonhandle ?method? ?channel? ?parameterdict?
j.request(?method=?,?channel=?,?parameters=?)
```

Assemble a properly formatted **JSON-RPC** request and potentially send it. The formatted message is returned as command result, and also sent via the channel if either that argument is a valid **Tcl** or **Python** channel handle, or the object has an associated channel in the internal attribute set.

All parameters are taken from the internal object attribute set if they are not explicitly set in the arguments. If arguments are supplied, the corresponding object attributes are also updated. Any arguments must be properly formatted - the method name is a simple string, and the parameters are a dictionary.

The internal message ID attribute of the object is incremented when this command is run, and the new value is transmitted. The first message ID of a newly created **JSON** object is one. To send a message without an ID, use the **`json send`** command.

json respond

```
json respond jsonhandle ?message? ?channel? ?id?
j.respond(?message=?,?channel=?,?id=?)
```

Assemble and potentially send a properly formatted **JSON-RPC** result message. The formatted message is stored in the *result* attribute of the **JSON** object, returned as command result, and also sent via the channel if either the argument is a valid **Tcl** or **Python** channel handle, or the object has an associated channel in the internal attribute set.

All parameters are taken from the internal object attribute set if they are not explicitly set in the arguments. If arguments are supplied, the corresponding object attributes are also updated. Any arguments must be properly formatted - the message argument is a **JSON** object encoding and the ID is an integer.

Since this command is a reply to a specific message ID, the internal object ID attribute is not incremented as in the **`json request`** command.

The command can also be spelled as **`json reply`**.

json send

```
json send jsonhandle ?method? ?channel? ?parameterdict?
j.send(?method=?,?channel=?,?parameters=?)
```

This command is essentially the same as **`json request`**, except that this is a notification for which no response is expected. The ID value in the message text is therefore always **`NULL`**, and internal ID attribute of the object is not incremented.

json set

```
json set jsonhandle ?attribute value?...
json set jsonhjandle dict
j.set(?attribute,value?,...)
j.set(dict)
j.set(?attribute=value?,...)
j.attribute = value
j[attribute] = value
```

Set one or more attributes of a **JSON** object. Since this paragraph is also referenced from the **`json get`** subcommand, the attribute set listed here includes attributes which cannot be set, or for which setting them to a scripted value does not usually make sense.

The currently supported set of attributes is:

- *body*
  The body part of the last received or sent **JSON-RPC** message. This attribute is not usually set.

- *bodylength*
  The length of the body in bytes. This is a read-only attribute.

- *channel*
  The communication channel associated with the **JSON** object. This is a standard **Tcl** channel handle. It is possible to set it to an empty string, which indicates that no channel is set.

- *errorcode*
  The error code from the last **JSON** parsing step as integer value. If no error occurred, the value is 0.
- *errormsg*
  The message of a **JSON** error, either as extracted from a **JSON-RPC** message, or set in preparation of sending an error message. If set, it must be formatted as a simple string.
- *errordata*
  Auxiliary informative data in an **JSON** error message, as per the **JSON-RPC** specification. This information is either extracted from a **JSON** message, or set in preparation for sending an error report. If set, it must be a properly formatted **JSON** object.
- handle
  A read-only attribute to get the **Tcl** handle of the object.
- *header*
  The header part of the last received **JSON-RPC** message. This attribute is not usually set.
- *headerlength*
  The length of the header in bytes. This is a read-only attribute.
- *host*
  The host the **JSON** object communicates with.
- *id*
  The **JSON-RPC** message ID. For sent messages, this is automatically incremented. If a received message contains a **NULL** ID, the value is set to zero.
- *method*
  The method name extracted either from the last received **JSON-RPC** message, or set in preparing to transmit a message.
- *parameters*
  The **JSON-RPC** method parameters. If queried, this is a dictionary of name/value pairs. In order to set these, a properly formatted dictionary must be supplied. The formatting is the same as for the *parameters* attribute of property definitions.
- *port*
  The port the **JSON** object uses for network communication. The default is 80, the standard **HTTP** port.
- *result*
  The method invocation result data extracted from the last received **JSON-RPC** message, or the data which is sent with the next response. If set, its format must be a properly formed **JSON** object. For further analysis, this value is usually decoded with a **`decode -json`** statement.
- *uri*
  The **URI** associated with a **JSON-RPC** service.

json subcommands

```
json subcommands
```

```
dir(Json)
```

This command returns a list of all the defined subcommands of the `json` command.

The keyx Command

This command is deprecated and intentionally undocumented.

The knode Command

The **knode** command manages **KNIME** nodes implemented as Cactvs scripts.

knode addparam

```
knode addparam khandle dialogtype name ?attribute value?..
knode addparam khandle dialogtype name attributedict
k.addparam(dialog,name,?attributedict?)
k.addparam(dialog,name,attribute=value,...)
```

Add a user-configurable parameter to the node definition. The first argument after the handle is the type of interface widget which should be used on the **KNIME** node configuration panel. It can be one of:

- *boolean*
  The parameter is a boolean value and can be set or unset via a checkbox.
- *columnselector* (or *colselector*)
  The parameter is a string which corresponds to a column name of a connected data table of the associated port.
- *colorchooser*
  The parameter is a color specification which can be selected via a color chooser widget.
- *diropen*
  The parameter is the name of a directory from which data will be read. This currently does only work if the node processor and a **KNIME** workbench share the file system.
- *dirsave*
  The parameter is the name of a directory from where data will be written. This currently does only work if the node processor and a **KNIME** workbench share the file system.
- *double*
  The parameter is a floating point value which can be set via a text line.
- *doublerange*
  The parameter is a floating point value pair. Its lower and upper boundaries can be set via text line widgets.
- *doublespinner*
  The parameter is a floating point value and can be adjusted via a spinner widget.
- *fileopen*
  |The parameter is the name of a file opened for reading via a file selector widget. Its contents are either transported via **RPC** to the node processor, or, if it has been indicated that the node server and the **KNIME** workbench share a file system, can be directly read by the processing script.

- *filesave*
  The parameter is the name of a file opened for writing via a file selector widget. The file name refers to a name in the file system visible to the **KNIME** workbench and is not necessarily directly accessible by the node processor. Instead of the specified remote file name, the node server sees the name of a local temp file it should use to save its data if it has not been indicated that workbench and node server share the file system. When processing has finished, these files are automatically sent to the remote workbench client via RPC and stored there under the originally specified name. The temporary file on the server side is deleted.
- *int*
  The parameter is an integer which can be set via a text line.
- *intspinner*
  The parameter is an integer and can be adjusted via a spinner widget.
- *label*
  The parameter has no value and cannot be modified, but, by setting a label text attribute, it still is shown on the configuration panel as text annotation.
- *multiline*
  The parameter is a string which can be input via a multi-line text box.
- *multistringselector*
  The parameter is a list of strings which can be selected from a set of predefined strings.
- *none*
  The parameter is not configurable via an interface widget. It still has a (string) value which can be queried in the processing script.
- *password*
  The parameter is a string which can be set via a text line where the content is hidden.
- *radiobutton*
  The parameter is a string which can be switched between various predefined values via a radio button widget.
- *string*
  The parameter is a string which can be set via a text line.
- *stringselector*
  The parameter is a string which can be chosen from various predefined strings.

The next argument is the name of the parameter. It is used to retrieve the parameter value or other attributes via the **`knode param`** command. It must be unique within a node. The maximum number of parameters for a node is 128. After configuration, querying the value attribute of a parameter by means of **`knode param`** returns the default value. Node that you cannot explicitly configure the *value* attribute.

The rest of the arguments are processed after the two mandatory initial arguments have been parsed. The additional attributes which can be set for a parameter depend on its data type and dialog widget class. These are all recognized attributes:

- *addnone*
  If set, add a *none* option to the selection. This attribute is unique to the column selector widget. If this option is chosen from the interface widget, an empty table column name is reported.
- *allowempty*
  If set, an empty string is an allowable parameter value. The attribute is value only for *string* and *multiline* dialog widgets.
- *columntypes* (or *coltypes*)
  A list of the cell data type classes which can be selected from an input table by a column selector widget. Possible values are *any* (the default), *structure* (any chemistry structure data cell type which encodes a complete structure, excludes *smarts*), *reaction* (any reaction data cell type), *number* (integer, long or double), *boolean*, *int*, *long*, *double*, *complex*, *string*, *xml*, *bitvector*, *blob*, *image*, *png*, *sdf*, *mol2*, *sln*, *smiles*, *smarts*, *mol*, *ctab*, *pdb*, *cml*, *inchi*, *cdxml*, *rxn*, *mrv*, *structure1d* (any structure notation without 2D or 3D coordinates, such as **SMILES**), *structure2d* (any structure cell type which can and is expected to store 2D coordinates), *structure3d* (any structure cell type which can and is expected to store 3D coordinates), *structureorstring* (any structure cell or a string cell assumed to hold structure data), *simple* (elementary data type, can be int, long, double, boolean or string), *simpleorstructure1d* (simple data or structure line notation) or *substructure* (any chemistry structure data cell type which can encode a substructure, includes *smarts* cells).
- *default*
  The default value which is preset and reported if the user configures nothing. The specified value must be parseable according to the parameter data type. For the *doublerange* type, the default must be a list of one or double floating point values. If only one value is given, it applies both to the lower and upper bound. For numerical values, an unspecified default is equivalent to a zero default value, and for strings an empty string.
- *description*
  A free-form parameter description text. It is used in the automatically written node documentation.
- *editable*
  A boolean flag indicating whether this value is user-editable. It only applies to string selector interface widgets. If set, the user can override the default choice set and enter a custom string instead.
- *extensions*
  A list of file extensions (with the dot) which is used to filter displayed file lists. Only supported for file and directory input and output dialog widgets.
- *group*
  The name of a group the dialog element is a member of. All dialog widgets on the same tab which share a common named group are laid out in a box with a frame. An empty string as name (the default) indicates that the dialog widget is not part of group.
- *height*
  The height of the dialog widgets in text lines. It only applies to *multiline* and *multistringselector* dialog widgets.

- *horizontal*
  If set, the dialog widget is laid out in horizontal direction from the previous widget. By default, widgets are laid out top to bottom.
- *label*
  A text label displayed next to the interactive widget.
- *max*
  The maximum allowed value. Applicable to the *integerbounded* and *doublebounded* settings types.
- *min*
  The *minimum* allowed value. Applicable to the *integerbounded* and *doublebounded* settings types.
- *name*
  The parameter name. Should not be set this way.
- *portassociation*
  The input port index or input port name a column selected is associated with. The default is 0 for the first input port.
- *property*
  A name of a toolkit property the parameter is related to. This is used for automatically generated node documentation.
- *required*
  A boolean flag whether a selection must be made by the user. Only applicable to column selectors.
- *settings*
  The settings Java class type used for this parameter. A suitable default settings class is automatically selected when the dialog type is set. In some cases, it may be useful to override it:
  - The *int* dialog defaults to the *integer* settings class. An alternative is *integerbounded*, which enforces minimum and maximum values.
  - The *double* dialog defaults to the *double* settings class, An alternative is *doublebounded*, which enforces minimum and maximum values.
  - The *intspinner* dialog default to the *integerbounded* settings class. An alternative is *integer*, which does not enforce minimum and maximum values.
  - The *doublespinner* dialog default to the *doublebounded* settings class. An alternative is *double*, which does not enforce minimum and maximum values.
- *step*
  The step size for the *intspinner* and *doublespinner* dialogs.
- *stringset*
  A list of strings which define choice options. This is used by *stringselector*, *multistringselector* and *radiobutton* dialogs.

- *stringset2*
  A list of strings which defined choice options. Currently unused, reserved for *columnfilter* dialogs which are not yet supported.
- *tab*
  The name of an additional tab in the configuration panel. Widgets for parameters without a tab name are placed on the primary tab. All parameters which share a common tab name are placed on an additional named tab in the order of definition. Multiple additional tab panes may be specified.
- *tablecolname*
  A table column name associated with the parameter. Used primarily for internal purposes. This is not the selected table column configured by a column selector widget - that is provided as *value* attribute like all other configuration values.
- *tooltip*
  A string displayed at the configuration widget if the mouse pointer hovers over it.
- *width*
  The widget width in characters. Applicable to *int*, *double*, *intspinner*, *doublespinner*, *string*, *password* and *multiline* widgets.

The command returns the current number of parameters on the node.

knode addport

```
knode addport khandle direction ?name? ?attribute value?...
knode addport khandle direction name attributedict
k.addport(direction,?name?,?attributedict?)
```

Add a port to the node definition. The first argument is either *input* or *output* and defines the port direction. The second argument it the port name. If the name is omitted, a name following the schema *input0...n* or *output0..n* is automatically generated. Without additional attributes, the port class is *datatable*. The maximum number of ports for a node is 16.

Additional attributes may me specified as attribute/value pairs or as a dictionary. The following attributes are recognized:

- *class* (or *type*)
  The port class. It can be either *datatable* (the default), *network*/*graph* or *image*. Other **KNIME** port types are currently unsupported. If this attribute is set, it should be the first in the attribute list because it influences the applicability or interpretation of other attributes.
- *description*
  A free-form test describing the role of the port. This is used for the automatically written node documentation which can be viewed for connected node in a **KNIME** workspace.
- *imagename*
  The name of the port image object. It is of little consequence for toolkit data processing, but is shown when the data ports of a connected node are viewed in the **KNIME** node menu. This attribute is only applicable to image ports.

- *mimetype*
  The mime type of the port data encoding. It is only used for image ports and identifies the image format. Typical values are *image/png*, *image/gif* or *image/svg+xml*. When result image data is set for a port, its encoding must match the pre-configured mime type.

- *optional*
  If set, **KNIME** workbench connections to this (input) port are optional. The node can be executed from the workbench with or without active connections to the port. When a table or network port object handle is queried for an unconnected optional port (see **`knode port`** command), an empty string is reported instead of the object handle.

- *networkname*
  The name of the port network object. It is of little consequence for toolkit data processing, but is shown when the data ports of a connected node are viewed in the **KNIME** node menu. This attribute is only applicable to graph ports, and it is not the same as a network object handle.

- *streaming*
  A boolean flag indicating if this port is streaming. The attribute has an effect only for data table ports. If it is not set (the default), all input table data is transmitted from a connected **KNIME** workspace before execution, and all result table data is sent after the execution has finished. In this model, the full input table content is immediately available, and previously added output table rows can be inspected. The disadvantage of this model is that all cell data needs to be kept in memory at the same time.

  With streaming data table ports, only a partial input row block has been sent when execution starts, and further data transfer overlaps with processing. Execution scripts usually remove input table rows from the top of the table (see **`table pop`** or **`table poploop`** commands) to keep it from growing. Determining whether all table data has been sent is possible by checking the *eod* table attribute. The same flag is internally tested by the standard table row loop commands.

  Rows added to streaming output tables are regularly sent back to the **KNIME** workspace and removed thereafter. There is no guarantee that a previously added output row can be revisited before it is sent and disappears. Execution scripts must be careful to add each output row and associated row information with a single, automatically synchronized statement. The advantage of streaming ports is that only a small part of the full table data needs to be kept in memory at any time. They can also be faster to run because transfer and computation are performed in concurrent threads. It is possible to mix streaming and non-streaming ports in a single node in arbitrary combinations.

- *tablename*
  The name of the port data table. It is of little consequence for toolkit data processing, but is shown when the data ports of a connected node are viewed in the **KNIME** node menu. This attribute is only applicable to data table ports and it is not the same as a table object handle.

knode bglisten

```
knode bglisten ?port?...
Knode.Bglisten(?port?,...)
```

Add one or more background **RPC** communication listener threads. If no port is specified, a lister on the standard port is configured. It can be examined and set via control variable *::cactvs(knimenode_default_port)*, its default is 16570.

An interpreter with an active background listener can receive and process **RPC** commands, for example from a **KNIME** workbench.

It is currently not possible to process **PYTHON** node scripts in background threads due to **PYTHON** multi-threading limitations.

The command has no return value.

knode bgunlisten

```
knode bgunlisten ?port?...
Knode.Bgunlisten(?port?,...)
```

Cancel listener threads active on specific ports. If no port is specified the standard port is assumed. The standard port can be examined and set via control variable *::cactvs(knimenode_default_port)*, its default is 16570.

Trying to cancel non-existing listener threads is silently ignored. If a thread is currently processing a **RPC** request, the cancellation happens after the current request has been served.

It is currently not possible to process **PYTHON** node scripts in background threads due to **PYTHON** multi-threading limitations.

The command has no return value.

knode clearparameters

```
knode clearparameters khandle
k.clearparameters()
```

Delete all currently defined parameters from the node.

The command can be shorted to **`knode clearparams`**.

The return value is the original node handle or reference.

knode clearports

```
knode clearports khandle
k.clearports()
```

Delete all currently defined ports from the node.

The return value is the original node handle or reference.

knode compile

```
knode compile khandle ?jarfile?
k.compile(?jarfile=?)
```

Compile the node definition into a **KNIME** workspace node. The result is a **JAR** file which can be copied into the drop-in directory of a local **KNIME** installation. By default, the name of the **JAR** file is constructed from the vendor domain, node name (or explicit Java class name, if set) and version, but it is also possible to request a custom **JAR** file name. A typical auto-generated **JAR** file name looks like *com.xemistry.ExcelSaver_1.1.jar*. After a **KNIME** program restart, the new node automatically shows up in the node browser.

The **JAR** file contains **RPC** interface code for communication with a node execution server, a normal-looking **KNIME** workspace node with ports, a configuration panel automatically assembled from the defined configuration parameters , auto-generated node documentation, node browser location and icon data, and the node definition code. When a node in a **KNIME** workspace connects to a node server, the node definition is transmitted and the node instantiated on the server. This node is then executed in the configuration and execution phases in response to normal **KNIME** workflow actions. Parameter configuration information is extracted from the configuration panel and sent in the configuration phase, together with column definition information of data cell ports. Port contents are either sent before execution, or, in streaming mode, parallel to execution. Likewise, result data from the node server is sent back either when the node execution has finished, or streamed in parallel to its execution.

Within the scripted node environment, data table port data appear as toolkit tables, graph port data as port objects, and image data as byte blob.

KNIME can be finicky about Java compiler versions. It may be necessary to configure the *javac* attribute so that the right compiler release is picked up.

knode configure

```
knode configure khandle ?paramname paramvalue?...
knode configure khandle paramdict
k.configure(?paramname,paramvalue?,...)
k.configure(paramdict)
```

Invoke the configuration script of the node, optionally with a custom set of configuration options. The *value* attribute of the named parameters are persistently changed to the new values, if this is allowable with respect to parameter type, minimum/maximum values, etc. If the setting of a parameter fails, an error is reported.

This command is typically used during node development, without a **KNIME** workspace connection. If a node is connected to a **KNIME** workspace, its configuration procedure is automatically invoked at the proper times.

Note that this command does not set input port data, which is at least partially defined when connected to a **KNIME** workspace via the output ports of connected nodes. During debugging outside of **KNIME**, suitable input port data for processing must be provided by script commands which either set the data explicitly, (see **`knode port`** command) or by setting input port filenames. In the latter case, their content is automatically loaded and transfered to the input ports before the configuration code is run.

If the configuration succeeds, the node is ready for execution by **`knode exec`** in debug mode.

The return value of the command is a list of the table handles or references, network handles or references and image bytes on the input ports in port index order.

`knode config` is a command alias.

knode create

```
knode create ?attribute value?...
knode create ?attributedict?
Knode(?attribute,value?,...)
Knode(attributedict)
Knode.Create(?attribute,value?,...)
Knode.Create(attributedict)
```

Create a new **KNIME** node object. Without any arguments an unconfigured default object is created. Additional arguments can be used to set attributes. The recognized attributes are the same as for the **`knode set`** command.

The return value is the handle or reference of the new **KNIME** node object.

knode delete

```
knode delete ?khandle?...
knode delete all
k.delete()
Knode.Delete(?khandle/kref?,...)
Knode.Delete("all")
```

Delete specific or all **KNIME** node objects from the toolkit interpreter instance. The return value is the number of successfully deleted **KNIME** node objects.

In case the node is actively connected with a **KNIME** workspace, the connection is cut and a network error is reported in the workspace. Generally, nodes actively connected to a workspace should only be deleted from there, which is performed by **RPC** communication from the workspace to the toolkit interpreter. This command is still useful for example in node development and compilation environments.

knode execute

```
knode execute khandle
k.execute()
```

Execute a **KNIME** node by invoking its execution procedure. It is assumed that the node has been successfully configured (see **`knode configure`** command).

This command is typically used during node development, without a **KNIME** workspace connection. If a node is connected to a **KNIME** workspace, its execution procedure is automatically invoked at the proper times.

If the execution succeeds, output port result data may be accessed by **`knode port`** commands.

The return value of the command is a list of the table handles, network handles and image bytes on all the output ports in port index order.

The command may be abbreviate to **`knode exec`**.

knode exists

```
knode exists khandle
k.exists()
Knode.Exists(handle/ref)
```

Check whether a node handle is valid.

knode get

```
knode get khandle attribute
k.get(attribute)
k.attribute
k[attribute]
```

Retrieve the current value of a node object attribute. All attributes which can be set can also be read (see **`knode set`** command). In addition, there are a couple of read-only attributes:

- *errorflag*
  A flag indicating the status of the last invocation of the configuration or execution procedure of this node. The flag is reset before any of these functions are called.
- *errormessage*
  The error message reported by the last invocation of the configuration or execution procedure of this node. The message is reset before any of these functions are called, and it is empty if there was no error.
- *inputportcount*
  The number of input ports on this node.
- *inputports*
  A list of the names of all input ports, in port index order.
- *inputtables*
  A list of the table handles or references of all input ports which are table ports, in port index order. If a port is not a data table port, or its table has not yet been configured, an empty string is reported for that port.
- *inrpc*
  A flag indicating weather the current commands are executed in the context of an **RPC** operation, or by a locally run script or interactive commands.
- *issafe*
  A flag indicating whether the current interpreter is safe (restricted) or not. By default, node interpreters are restricted node-specific slaves of the main interpreter.
- *handle*
  The handle of the current node. This is useful in case the node is addressed via its instance **UUID**.
- *lastrun*
  The time stamp of the last time the execution function of the node was called.
- *lastruntime*
  The number of seconds spent running the execution function of the node the last time.
- *log*
  A list of all log messages accumulated in the last run. Every list element is a nested list with the log level and the log message text.
- *nodetype*
  The general type of the node (*source*, *sink*, *manipulator* or *other*).

- *outputportcount*
  The number of output ports on this node.
- *outputports*
  A list of the names of all output ports, in port index order.
- *outputtables*
  A list of the table handles or references of all output ports which are table ports, in port index order. If a port is not a data table port, or its table has not yet been configured, an empty string is reported for that port.
- *parameters*
  A nested list of the setting of all node configuration parameters. Every list element is a dictionary with all applicable attributes of a parameter. Its exact content is dependent on the dialog widget configured for the parameter. Dictionary entries supported for all parameter types are *description, dialog, label, group, horizontal, name, property, settings, tab, tooltip, tablecolname* and *value*. Other dialog-dependent entries are *addnone, allowempty, columntypes, default, editable, extensions, height, min, max, portassociation, required, step, stringset* and *width*.
- *params*
  A short alias for *parameters*.
- *ports*
  A nest list with the most relevant general port information. Every list element contains the port name, its direction (*input* or *output*), its port class (*datatable*, *image* or *network*), the port description string, the *streaming* flag, the *optional* flag, the *configreset* flag, the port table name, and the port table handle, if it is a data table port and it is already configured.
- *scriptlanguage*
  The language used in the configuration and execution script blocks. It is either **Tcl** or **Python**. Its type is implicitly defined by which commands the script source is set.
- *tmpdir*
  The temporary directory to be used by this node. Depending on the server configuration, this can be either the normal system directory, or a specially protected node-specific directory. Nodes should not assume that they have access to the system *tmp* directory, or any other file-system location, except this location.
- *totalprocessedinrows*
  The total number of rows in input tables this node instance has so far received from a connected **KNIME** workbench..
- *totalprocessedoutrows*
  The total number of output table rows this node instance has so far sent to a connected **KNIME** workbench.
- *totalruntime*
  The total execution time this node instance has accumulated so far.

knode list

```
knode list ?pattern?
```

```
Knode.List(?pattern=?)
```

Return a list of all currently existing **KNIME** object handles. If a filter pattern is specified, only handles matching it are listed. The pattern syntax is the same as in the standard **TCL** command **`string match`**.

knode listen

```
knode listen ?port?
Knode.Listen(?port=?)
```

Start a foreground **RPC** listener. If no port is specified, the standard port is assumed. It can be examined and set via control variable *::cactvs(knimenode_default_port)*, the default is 16570.

This command does not return except on error or the listener loop has been canceled by outside actions.

knode log

```
knode log khandle ?level? message
k.log(?level=?,?message=?)
k.log(message)
```

Add a log message to the node state. The level may be one of *debug*, *info*, *warning*, *error* or *fatal*. The default level is *info*. The message text is a free-form string. If the message is empty, no actual log entry is written.

Log messages are transmitted to a connected **KNIME** workspace and show up in the console window.

The command returns the original node handle or reference.

knode parameter

```
knode parameter khandle parametername ?attribute?
k.parameter(name=,?attribute=?)
```

Retrieve an attribute or the current value of a node configuration parameter. Within the context of a live **KNIME** workbench connection, these are automatically extracted from the configuration panel settings and available both in the configuration and execution procedures.

The current parameter values (but not other attributes) can also be accessed from within a node configuration or execution script as global array variable **`::params`**.

If no attribute is specified, the parameter value is retrieved.

Parameters may be accessed via the parameter name or the 0-based parameter index.

`knode getparam`, **`knode getparameter`** and **`knode param`** are command aliases.

The following attributes are recognized:

- *addnone*
  A boolean flag indicating whether the parameter dialog contains an additional *none* option.
- *allowempty*
  A boolean flag indicating whether the parameter dialog field may be left unfilled.

- *columntypes* (or *coltypes*)
  A list of native **KNIME** table column types a column selector may be applied to.
- *default*
  The default value of this parameter.
- *description*
  A free-form descriptive string.
- *dialog*
  The type of dialog widget associated with this parameter in the configuration panel of the Java interface class.
- *editable*
  A boolean flag indicating whether the value is editable in the configuration panel.
- *extensions*
  A list of allowed or display-filtered file extensions for file I/O selector widgets.
- *filecontent*
  The file bytes from a file input selector widget. For this widget, the *value* attribute returns the file name. If no file was specified, an empty result is returned.
- *filecontentlen*
  The length of the file content from a file input selector widget in bytes. If no input file was specified, or it could not be opened, the attribute value is minus one. This allows distinction between empty, but readable input files and unspecified or unreadable files.
- *group*
  The name of the widget group this parameter is part of, or an empty string if it is not. Widget groups are rendered within a frame in the **KNIME** configuration panel.
- *horizontal*
  A boolean flag. If set, the layout of the dialog widget is performed in horizontal and not vertical direction.
- *index*
  The 0-based parameter index value.
- *label*
  The text label used to name the widget in the configuration panel.
- *max*
  The maximum value of the parameter. Only defined for bounded integer and double settings.
- *min*
  The minimum value of the parameter. Only defined for bounded integer and double settings.
- *name*
  The parameter name. Useful if the parameter was identified via its index.
- *portassociation* (or *port*)
  For column selector widgets, the input port index this selector applies to.

- *property*
  The name of a toolkit property this parameter is associated with. If there is no association, this is an empty string. This attribute is used in the auto-generated node documentation.
- *required*
  A boolean flag indicating whether this parameter must have been set by the user.
- *settings*
  The enumerated class name of the *settings* object associated with the parameter. This implicitly defines the datatype of the parameter and is interdependent with the associated dialog type.
- *stringset*
  A list of strings, for example defining possible values of a string selector widget.
- *stringset2*
  A second list of strings. Used only in column filter dialogs.
- *tab*
  The name of an additional tab in the configuration panel the widget for this parameter is placed on. Parameters on the default tab report an empty tab name.
- *tablecolname* (or *colname*)
  The name of a table column selected by a column selector widget.
- *tooltip*
  The tooltip string displayed by the KNIME configuration panel if the user hovers with the mouse over a configuration widget.
- *value*
  By far the most important attribute - this is the currently configured value of the parameter.

knode port

```
knode port khandle portname attribute
knode port khandle portname table ?tablehandle|#new? ?isowner?
knode port khandle portname imageblob ?imagebytes?
knode port khandle portname imagefile ?filename?
knode port khandle portname network ?networkhandle|#new? ?isowner?
knode port khandle portname expectedrows ?count?
k.port(name=,attribute=,?value=?,?isowner=?)
```

Retrieve information about a a specific port, or set it. Ports can be identified by their name, or by their 0-based port index. Some port attributes can be modified. In this case, one or two additional parameters for the new attribute value follow the attribute name argument.

The following attributes are recognized:

- *buffersize*
  The buffer size of the port.
- *class*
  The port class (*datatable*, *image* or *network*).

- *configreset*
  If set to 0, the output port table configuration is not reset before the configuration procedure. By default, all tables associated with output ports are deleted before the function is called.
- *description*
  Free-format description string.
- *direction*
  Port direction (*input* or *output*).
- *expectedrows*
  The number of table rows this port is expected to receive until end-of-data. For input ports, this information is transmitted via **RPC** from the **KNIME** workbench in case the size of the input table is known. The value can be set, typically from the configuration procedure, and this influences for example automatic node execution progress reporting in some progress reporting modes.
- *filename*
  Specify the name of a file which contains data to be loaded into an input port, or the name of a file to which port contents should be written after execution for output ports. This is either a table data file (for data table ports, usually in native **KNIME** table format), a network data file (for network ports, usually in native **KNIME** table format), or a image for for image ports. The purpose of this attribute is to facilitate node debugging outside of **KNIME**. Input ports are loaded when the **`node configure`** command is executed, but not when a node is configured from within a server listener thread. Likewise, port contents are written when a **`node exec`** command has successfully completed execution, but again not when the execution was performed in a server thread.
- *imageblob*
  On retrieval, this returns the currently set image bytes for the port, which is defined only for image ports. This attribute can be set on image ports with data bytes representing a **GIF**, **PNG** or **SVG** image. The image content is transmitted back to the **KNIME** workspace when the node execution has finished.
- *imagefile*
  On retrieval, this returns the name of a temporary file filled with the currently configured image bytes for this port, which is defined only for image ports. This attribute can be set on image ports with the name of a file storing a **GIF**, **PNG** or **SVG** image. The file bytes are read and stored as image bytes. The name of the input file is not remembered. The image content is transmitted back to the **KNIME** workspace when the node execution has finished
- *imagename*
  The name (not handle or reference) of the image associated with this port, as shown in the port menu on nodes in a **KNIME** workspace.
- *index*
  The port index. The port index is assigned in the order of port definitions (see **`knode addport`** command).
- *mimetype*
  The mime type associated with the port. This is useful only for image ports.

- *name*
  The port name.
- *optional*
  A boolean flag indicating whether this port may remain unconnected in the **KNIME** workspace. In configuration procedures, unconnected optional input ports do not possess an associated automatically set up table or network object, or image bytes.
- *network*
  On retrieval, return the handle or reference of the network object associated with the port, or an empty string if no such network exists. If the port is a non-optional or output network port, and no network object has yet been configured, retrieval results in implicit creation of an empty network object and its association with the port. This attribute may be set for network ports. The argument is either the handle of an existing network object, an empty (also `None` for **PYTHON**) string (which destroys any existing network on the port) or the magical string *#new*, which creates a new empty network and associates it with the port. After association, the network life-cycle is managed by the node and cannot be deleted via its handle. When the node is destroyed, or the network replaced, it is automatically deleted, except when the optional *isowner* flag is also specified when the port network is set. With this flag set, the script writer remains responsible to delete the network when it is no longer needed, but this can be done only after the node association has been broken.
- *networkname*
  The name (not handle or reference) of the network associated with this port, as shown in the port menu on nodes in a **KNIME** workspace.
- *processedrows*
  The number of rows which have been received from or sent to a connected KNIME workspace, via this port during the current node execution run.
- *streaming*
  A boolean flag indicating that transfer of input table rows, or sending of result table rows, should proceed in parallel with node execution for this port. If the flag is not set, input tables are fully filled when node execution begins, and output tables are only sent after node execution finishes. Otherwise, these operations overlap. If streaming is active, specific node execution code patterns must be followed. Streaming can only be used on data table ports.
- *table*
  On retrieval, return the handle or reference of the table object associated with the port, or an empty string if no such table exists. If the port is a non-optional or output data table port, and no table object has yet been configured, retrieval results in implicit creation of an empty table object and its association with the port. This attribute may only be set for data table ports. The argument is either the handle of an existing table object, an empty (or `None` for **PYTHON**) string (which destroys or dissociates any existing table on the port) or the magical string *#new*, which creates a new empty table and associates it with the port. After association, the table life-cycle is managed by the node and cannot be deleted via its handle. When the node is destroyed, or the table replaced, it is automatically deleted, except when the optional *isowner* flag is also specified when the port table is set. With this flag set, the script writer remains responsible to delete the table when it is no longer needed, but this can be done only after the node association has been broken.

- *tablename*
  The name (not handle) of the table associated with this port, as shown in the port menu on nodes in a **KNIME** workspace.

knode read

```
knode read filename
Knode.Read(filename)
```

Create a node by reading an **XML**-based node definition file. The default node file suffix is *.knd* (**KNIME** node definition), but this is not enforced.

The return value is the handle or reference of the new node.

knode readblob

```
knode readblob blobdata
Knode.Readblob(blobdata)
```

Create a new by reading an **XML**-based node definition file directly from string data.

The return value is the handle or reference of the new node.

knode reset

```
knode reset khandle
k.reset()
```

Reset the node so that its state is the same as after a **`knode create`** command. All result data, port and parameter definitions as well as accumulated log messages are deleted and the basic node attributes are reset to default values.

The command returns the original node handle or reference.

knode resetlog

```
knode resetlog khandle
k.resetlog()
```

Reset the accumulated log information of the node. When connected to a **KNIME** workspace, log messages which have been transferred to the workspace are automatically deleted. Also, any new call to the node configuration or execution functions resets the log.

The command returns the original node handle or reference.

knode set

```
knode set khandle ?attribute value?...
knode set khandle dict
k.set(?attribute,value?,...)
k.set(dict)
k.attribute = value
k[attribute] = value
```

Set zero or more node object attributes. The recognized attributes are:

- *adaptercells*
  A boolean flag indicating whether the node should use the new multi-encoding Adapter cell **KNIME** table cell classes for storing structure data. If set, structure data cells sent back to a **KNIME** workspace will be automatically backed up by a native toolkit version of the structure, which makes decoding faster and retains computed structure data directly on the structure object. If a backing native toolkit encoding is present when a cell with structure data is decoded, it is used preferably over other encodings such as **SMILES** or **MDL** *Molfile* which might also be present. It is recommended to set this attribute if your **KNIME** installation is sufficiently recent.
- *address_city*
  The city part of the author contact address.
- *address_country*
  The country part of the author contact address, following the ISO3166 standard.
- *address_state*
  The state part of the author contact address. Empty if not applicable.
- *address_street*
  The street address part of the author contact address. Includes floor, house number, etc.
- *address_zip*
  The **ZIP** code or other applicable postal code of the author contact address.
- *affiliation*
  The institution the author of the node works for.
- *affiliationduns*
  The **DUNS** registration ID of the affiliated institution. This is primarily useful for US government projects.
- *affiliationurl*
  The **URL** of the affiliated institution.
- *author*
  The author of the node. This is a free-form string.
- *authorurl*
  A **URL** with information on the author, or an empty string if unset.
- *beta*
  A boolean flag indicating beta status of the node. If set, the node character is indicated in the auto-generated node documentation which can be viewed within a **KNIME** workspace.
- *cancelrequest*
  A boolean flag. If set, it indicates the desired to cancel the currently executing **RPC** command on the node. This is primarily useful for debugging and not used in normal programming. The cancellation request flag is automatically reset if the cancellation has been performed.
- *category*
  A category string to be used if the node definition is stored in a repository. The default category is *CactvsNodes*. This is the section under which in a **KNIME** workbench the node is listed after installation.

- *categorydescription*
  A free-form text describing the category defined above.
- *categoryiconfile*
  A **PNG** icon file associated with the node category. It should be size 16x16. The icon is displayed in the node browser of **KNIME** workbenches with the category name. The default icon is a small green flowering cactus.
- *classuuid*
  The base class **UUID** of this node definition.
- *comment*
  A free-form text comment.
- *configproc*
  The name of the **TCL** procedure which handles the node configuration phase as initiated from a **KNIME** workspace. The function is passed the handle of the node to configure as a single argument. The most important task of a *config* procedure is to set the columns of node output tables, usually after inspecting the settings of configuration options and the column definitions of connected input tables. The option and input table column definition data is available at configuration time, but not input table cell (or other input port) content. The default name for the configuration procedure is *config*.
- *configscript*
  A body of **TCL** code containing the procedures handling node configuration. Often this is just a single procedure. The name of one of the procedures defined in this block must match the value of the *configproc* attribute. This procedure is called as entry point. Other procedures defined in the block are visible, but otherwise the configuration script executes in a restricted slave interpreter which is instantiated on a per-node basis. *tclconfigscript* is an alias.
- *configscriptfile*
  The name of a file which contains the **TCL** source code of the configuration section. The file content is read and processed like the more commonly used in-line *configscript* attribute.
- *configtrace*
  A boolean flag which, if set, requests command tracing of the configuration procedure. This applies only to **TCL** code. The trace output is visible only on a local interpreter instance, not from a **KNIME** workbench.
- *date*
  This attribute stores the date of the node definition compilation. It is used for information purposes only. If a new node is compiled, it is initialized to the current date.

- *deadmantimeout*
  Set a node-specific deadman timeout. The attribute is an integer with milliseconds as unit. A non-zero value set here overrides a globally set general **KNIME** node deadman timeout. If the execution of the configuration or execution script on this node exceeds the timeout, the application server process is summarily killed. This feature is primarily used to protect node servers from scripts maliciously hogging resources. A killed node server process results in errors on all connected **KNIME** workspace clients, but nodes linked to a server can automatically re-connect if the server process is re-started (which is done automatically in a standard node server set-up), and node result data already transferred to a workspace is not compromised.
- *defaultlanguage*
  The default language for automatically generated set-up and test scripts in the absence of other indicators, such as suffixes in the names of output files. It can be either **Tcl** or **Python**.
- *deprecated*
  A boolean flag indicating deprecated status of the node. If set, the node character is indicated in the auto-generated node documentation which can be viewed within a **KNIME** workspace.
- *doi*
  A digital object identifier for the node, if defined.
- *editurl*
  The default node designer **URL** to export the node definition to via the configuration panel button of a node, if the node was compiled with an open definition. The default is *https://xemistry.com/nodedesigner*.
- *email*
  The email address of the creator of the node, useful in case a definition file is distributed and anybody has questions.
- *execproc*
  The name of the **Tcl** procedure which handles the node execution phase as initiated from a **KNIME** workspace. The function is passed the handle of the node to configure as a single argument. The function is called after a successful configuration phase. Usually, all input and output ports have already been set up, and at least an initial data portion is already present in the input tables or other input port objects.The execution procedure then reads the input table rows or other input data and, usually depending on configuration panel settings, appends output table rows. In case of streaming input tables, new input data continues to arrive while the procedure executes, and input rows are usually popped from the input tables during processing to avoid accumulating large amounts of table data. For streaming output tables, result rows which have been transmitted to a connected workspace are automatically removed..The default name for the execution procedure is *exec*.
- *execscript*
  A body of **Tcl** code containing the procedures handling node execution. Often this is just a single procedure. The name of one of the procedures defined in this block must match the value of the *execproc* attribute. This procedure is called as entry point. Other procedures defined in the block are visible, but otherwise the execution script executes in a restricted slave interpreter which is instantiated on a per-node basis. *tclexecscript* is an alias.

- *execscriptfile*
  The name of a file which contains the **Tcl** source code of the execution section. The file content is read and processed like the more commonly used in-line *execscript* attribute.
- *exectrace*
  A boolean flag which, if set, requests command tracing of the execution procedure. This applies only to **Tcl** code. The trace output is visible only on a local interpreter instance, not from a **KNIME** workbench.
- *expertflag*
  A boolean flag indicating expert status of the node. If set, the node character is indicated in the auto-generated node documentation which can be viewed within a **KNIME** workspace.
- *fork*
  A boolean flag indicating that the node should be configured and executed in a forked node server instance, which automatically terminates when execution is complete. For **Python**-based nodes, this is automatically enforced because of the severe multi-threading limit of **Python** interpreters and their dangerous lack of intra-interpreter isolation.
- *fulldescription*
  A long description text. It is displayed in the auto-generated node documentation which can be viewed in a **KNIME** workbench.
- *keywords*
  A free-form list of keywords associated with the node.
- *id*
  The official registration ID of the node. *regid* is an alias.
- *infourl*
  A **URL** with information on the node, or an empty string if unset.
- *instanceuuid*
  The **UUID** of this node instance. It is usually not recommended to set or change this by script commands, since it is essential for **RPC** communication.
- *javac*
  The path to the Java compiler to use for the compilation of the Java code auto-generated for the **KNIME**-side node interface object. If it is not set, an attempt is made to find a Java compiler in the normal executable path. Note that **KNIME** can be finicky about Java compiler version mismatch. Even if you find a *javac* by normal path search, it may not be the right one among multiple installation.
- *keepjavasource*
  If set, the Java source automatically generated for the **KNIME**-side node interface class is retained after compilation. This is a debug option.
- *knimeplugindir*
  The standard directory for user-defined nodes in the local **KNIME** installation. If is is set, newly compiled nodes are automatically copied to this location.

- *lenient*
  A boolean attribute. If it is set, the applicability of parameter attributes with respect to their dialog type is not checked. For normal scripting the flag should not be used, but it is helpful when setting up automatic processing of node definitions.
- *license*
  The license class associated with this node. Setting the license to a standard type updates the associated **URL** with a standard location.
- *licenseurl*
  A **URL** with details about the node license.
- *listcells*
  If set, this node may use list cells for data communication with a **KNIME** workbench, for example for array- and compound-class property data. List cells are only available in more recent **KNIME** versions.
- *literature*
  A free-form literature reference.
- *localfs*
  If set, this indicates that the **KNIME** workbench and the node server share a common file system. This means that file data from configuration panels or data cells does not need to be transferred via the **RPC** connection. Instead, file names can be sent, which makes processing more efficient for larger files.
- *mintoolkitversion*
  A version number in 1.2.3.4 or shortened notation indicating the minimum toolkit version of a node server which is able to execute this node, for example because the node uses features or properties which were introduced only in recent toolkit versions. If an older node server receives node code with a higher version requirement, execution is denied.
- *msglogfile*
  The name of a file which captures all messages generated by a node. Messages also appear in the console of a connected **KNIME** workbench. On servers, custom message file logs are usually not allowed, so this is primarily a debug feature for local node development. Magic file names *stdout* and *stderr* may be used, and an empty log file name disables per-node message logging. This attribute cannot be queried.
- *name*
  This is the name of the node. If this is changed, the node class root name and default JAR file name are also implicitly changed.
- *naptime*
  The number of microseconds to sleep in transfer threads if no data is available to be sent or received. The default are 25000 microseconds. A lower value decreases processor load, a higher value make nodes appear less responsive.
- *nodeclassroot*
  The root Java class name of the **KNIME** workbench interface object. This is usually not set directly, but auto-derived from the node name.

- *nodeiconfile*
  A file which contains a custom node icon as **PNG** file. The standard icon size are 16x16 pixels. This icon is displayed in the node browser of a **KNIME** workbench. The default icon is a small green flowering cactus.

- *orcid*
  The ORCID code of the author of the node (see www.orcid.org).

- *path*
  The menu path associated with the node.

- *phone*
  A contact phone number of the author.

- *pluginjarfile*
  The file name of the compiled Java node interface class for use in a **KNIME** installation. Usually, this is not set ad the **JAR** file name is auto-derived from the node name.

- *progressmessage*
  A free-form string which is used as component of node execution progress messages.

- *progressmode*
  The method to report node execution progress to a connected **KNIME** workbench. The default is *none*, and no progress updates are sent. Other supported modes are:

  - *scripted* - the progress indicator is updated from within the execution script.
  - *portrows* - the progress value is the percentage of input and output rows of all table ports which have been read or written vs. the expected total row count.
  - *porteods* - the progress value is the portion of all ports which have reached end-of-data status.
  - *inportrows* - like *portrows*, but computed only for input table ports.
  - *inporteods* - like *porteods*, but computed only for input ports.
  - *outportrows* - like *portrows*, but computed only for output table ports.
  - *outporteods* - like *porteods*, but computed only for output ports.
  - *inportrows0* - like *inportrows*, but computed only for input port 0.

  Modes *inportrows[0-2]*, *inporteods[0-2], outportrows[0-2]* and *outporteods[0-2]* follow the same schema.

- *property*
  The name of a single toolkit chemical data-related property which is computed by this node. This information is used for the auto-generation of node documentation. Setting this attribute appends a property to the node property register. It does not overwrite previous entries. The maximum size of the property register is 32 entries. Multiple properties can be set at once, or the register reset, with the *properties* attribute. This attribute cannot be queried.

- *properties*
  A list of toolkit chemical data-related properties which are computed by this node. This information is used for the auto-generation of node documentation. The maximum size of the property register is 32 entries.
- *pyconfigscript*
  Essentially the same as *tclconfigscript/configscript*, except that the configuration procedure block is expected to be a **Python** function collection, not a **Tcl** procedure block. Note that the use of configuration or execution scripts written in **Python** is currently limited, insecure and burdened with mandatory fork overhead compared to the **Tcl** counterpart.
- *pyconfigscriptfile*
  Essentially the same as *tclconfigscriptfile/configscriptfile*, except that the configuration source code file is expected to be a **Python** function collection, not a **Tcl** procedure source file. Note that the use of configuration or execution scripts written in **Python** is currently limited, insecure and burdened with mandatory fork overhead compared to the **Tcl** counterpart.
- *pyexecscript*
  Essentially the same as *tclexecscript/execscript*, except that the configuration procedure block is expected to be a **Python** function collection, not a **Tcl** procedure block. Note that the use of configuration or execution scripts written in **Python** is currently limited, insecure and burdened with mandatory fork overhead compared to the **Tcl** counterpart.
- *pyexecscriptfile*
  Essentially the same as *tclexecscriptfile/execscriptfile*, except that the execution source code file is expected to be a **Python** function collection, not a **Tcl** procedure source file. Note that the use of configuration or execution scripts written in **Python** is currently limited, insecure and burdened with mandatory fork overhead compared to the **Tcl** counterpart.
- *pyscript*
  This attribute allows the definition of both the configure script and the execution script in one blob of **Python** source code. See *pyconfigscript* and *pyexecscript* attributes.
- *pyscriptfile*
  This attribute allows the import of both the configure script and the execution script from one file with **Python** source code. See *pyconfigscriptfile* and *pyexecscriptfile* attributes.
- *renderproperty*
  The name of a single toolkit chemical rendering-related property which is computed by this node. This information is used for the auto-generation of node documentation. Setting this attribute appends a property to the node property register. It does not overwrite previous entries. The maximum size of the property register is 32 entries. Multiple properties can be set at once, or the register reset, with the *renderproperties* attribute. This attribute cannot be queried.
- *renderproperties*
  A list of toolkit chemical rendering-related properties which are computed by this node. This information is used for the auto-generation of node documentation. The maximum size of the property register is 32 entries.

- *rowblocksize*
  The number of table data rows which are transferred within one **RPC** message block, if that much data is available. The default is 50. Changing this parameter is not advised under normal circumstances.

- *rpcflags*
  A set of flags influencing the **RPC** communication between node server and **KNIME** workbench. Bit positions of this flag are usually set by dedicated script commands. Recognized flags include *none*, *localfs*, *secure* and *structureexport*.

- *rpchost*
  The name of the host this node communicates with. This is usually not set manually but transmitted as part of the credentials of a **KNIME** workbench node interface object.

- *rpclogfile*
  The name of a file which logs debug information about the **RPC** communication with a **KNIME** workbench. Magic file name *stdout* and *stderr* are supported, and an empty filename (or `None` for **PYTHON**) disables **RPC** logging. **RPC** logging on remote servers is usually disabled, so this is primarily a debug feature for local node development.

- *rpcpassword*
  The **RPC** password. This is usually not set manually but transmitted as part of the credentials of a **KNIME** workbench node interface object.

- *rpcport*
  The port for **RPC** communication. This is usually not set manually but transmitted as part of the credentials of a **KNIME** workbench node interface object.

- *rpcuser*
  The remote user of this node. This is usually not set manually but transmitted as part of the credentials of a **KNIME** workbench node interface object.

- *script*
  This attribute allows the definition of both the configure script and the execution script in a single blob of **TCL** source code. See *configscript* and *execscript* attributes.

- *scriptfile*
  This attribute allows the import of both the configure script and the execution script from a single file with **TCL** source code. See *configscriptfile* and *execscriptfile* attributes.

- *shortdescription*
  A free-form short node description string. It is used as part of the auto-generated node documentation which can be viewed in a **KNIME** workbench.

- *showxmlsource*
  If set, the compiled Java interface node contains an extra tab in the configuration panel which contains the source of the node definition in **XML** format (see `knode write` command). This text can be *copied&pasted* for easy sharing of the node source code and further third-party modification of the node.

- *structureexport*
  This flag can be set to values *never*, *parameterdependent* and *always*. If set to anything but *never*, the node may, or always does, send (potentially sensitive) structure information to 3rd party sites, for example for the lookup of structure data from Internet sites. If that is the case, the user must approve this operation by checking an automatically provided checkbox in the node configuration panel on the **KNIME** workbench. Otherwise, the node will not export structure data and fail.
- *tclconfigscript*
  See *configscript*, this is an alias.
- *tclconfigscriptfile*
  See *configscriptfile*, this is an alias.
- *tclexecscript*
  See *execscript*, this is an alias.
- *tclexecscriptfile*
  See *execscriptfile*, this is an alias.
- *tclscript*
  See *script*, this is an alias.
- *tclscriptfile*
  See *scriptfile*, this is an alias.
- *unrestricted*
  If this flag is set, the node requires an unrestricted **Tcl** interpreter for execution. with file system and network access. Whether a server honors this request, is a server configuration issue. Public servers will generally refuse execution of such nodes because this is a severe security risk. **Python** does not support restricted interpreters and such nodes are therefore always considered unrestricted (and not executed on public servers).
- *vendordomain*
  The domain of the vendor or developer of the node. This is used as part of the Java class configuration. The default is *xemistry.com*.
- *version*
  The version of the node. This is a string in a 1.2.3 (or shortened) style. If the version changes, the default **JAR** file name is automatically updated.
- *versionuuid*
  The version **UUID** associated with this node version.
- *workdir*
  The name of a directory for temporary files assembled and compiled during the assembly of a Java **JAR** file. The directory is created if it is not yet present. It must be writable. By default, a random directory in the system-dependent temporary files directory is created and automatically deleted after **JAR** file generation. If a custom work directory is used, it is not auto-deleted, but internal cleanup still proceeds except in case Java sources are retained (see *keepjavasource* attribute).

knode subcommands

```
knode subcommands
dir(Knode)
```

List all subcommands of the **knode** command.

knode write

```
knode write khandle ?filename?
k.write(?filename=?)
```

Write the node definition to an **XML**-based file which is suitable for reading with the **knode read** command.

If no filename is specified, its name is derived from the node name with a *.knd* (**KNIME** node definition) suffix. If an explicit empty filename is specified (including **None** for **PYTHON**), an in-memory string blob is produced, without writing to a file.

The definition file only saves the node definition with all attributes, ports and parameters, but no execution state or execution result information.

The return value is the file name, or, in case of blob output, the definition file contents.

knode writeconfig

```
knode writeconfig ?filename?
Knode.Writeconfig(?filename=?)
```

Write the current system configuration regarding the **KNIME** node subsystem to a file. If no file name is given, the default configuration filename *csconfig.xml* in the current directory is used.

Files of that name are automatically read on start-up if they are in the current working directory, or in a standard location such as the home directory or the **CACTVS** installation directory, or passed as a program option when the interpreter is started. This makes it easy to maintain a custom node execution environment without explicitly setting many custom configuration variables.

The return value is the file name.

knode writedoc

```
knode writedoc khandle ?filename?
k.writedoc(?filename=?)
```

Write a **HTML** file for a node which looks very much like the contents of the node description panel in the **KNIME** workbench. The original documentation compiled into the node **JAR** files is **XML**-coded, so this is not exactly the same data. The constant contents of the *RPC* and *Source* tabs are omitted.

If no node name is specified, a name is automatically constructed from the node name and a *.html* suffix. If an explicit empty filename is specified (including **None** for **Python**), the output is an in-memory string blob, not a file

The return value is the file name, except in case of blob output, when it is the **HTML** file contents.

knode writescript

```
knode writescript khandle ?filename?
```

```
k.writescript(?filename=?)
```

Write a **Tcl** or **Python** script which, when run, faithfully recreates the node definition and writes out a new **XML** node definition file and a new **KNIME JAR** file. Editing these scripts in the context of custom node development is much more convenient than manipulating the content of the native **XML**-based node definition file. For simple node source code distribution, the **XML** format is more suitable.

If the file name has a *.tcl* suffix, the output is a **Tcl** script. The script language is automatically switched to **Python** if the file name has a *.py* suffix. If none of these suffixes is used, the script language configured via the *defaultlanguage* node attribute is used. If that is also unset, **Tcl** is the default. If no file name argument is used, the file name is constructed from the node name and the appropriate language suffix. If an explicit empty filename is specified (including **None** for **Python**), the output is an in-memory string blob, not a file.

The return value is the file name, except in case of blob output, when it is the script file contents.

knode writetest

```
knode writetest khandle ?filename?
k.writetest(?filename=?)
```

Automatically generate a **Tcl** or **Python** test script for a node. The command line arguments for the script are port data file names for all non-optional input ports, optionally file names to store the output of ports, and a parameter dictionary. The script contains readable comments to self-document its use.

If the file name has a *.tcl* suffix, the output is a **Tcl** script. The script language is automatically switched to **Python** if the file name has a *.py* suffix. If none of these suffixes is used, the script language configured via the *defaultlanguage* node attribute is used. If that is also unset, **Tcl** is the default. If no file name argument is used, the file name is constructed from the node name with a _*test* name modification and the appropriate language suffix. If an explicit empty filename is specified (including **None** for **Python**), the output is an in-memory string blob, not a file.

The return value is the file name, except in case of blob output, when it is the script file contents.

The lhasa Command

The **`lhasa`** command manages LHASA objects. A LHASA object is able to read and utilize the original LHASA transform files as source of detailed reaction knowledge.

The command has the following subcommands:

lhasa append

```
lhasa append lhandle ?property value?...
l.append({?property:value,?...})
l.append(?property,value,?...)
```

Standard data manipulation command for appending property data. It is explained in more detail in the section about setting property data.

The command returns the first data value.

lhasa assign

```
lhasa assign lhandle srcprop dstprop
l.assign(srcproperty=,dstproperty=)
```

Assign property data to another property on the same Lhasa object. Both properties must be associated with the Lhasa object class. This process is more efficient than going through a pair of **`lhasa get/lhasa set`** commands, because in most cases no string or TCL/PYTHON script object representations of the property data need to be created.

Both source and destination properties may be addressed with field specifications. A data conversion path must exist between the data types of the involved properties. If any data conversion fails, the command fails. For example, it is possible to assign a string property to a numeric property - but only if all property values can be successfully converted to that numeric type. The reverse example case always succeeds, out-of-memory errors and similar global events excluded.

The original property data remains valid. The command variant **`lhasa rename`** directly exchanges the property name without any data duplication or conversion, if that is possible. In any case, the original property data is no longer present after the execution of this command variant.

Examples

```
lhasa assign $lh L_IDENT L_NAME
```

lhasa classify

```
lhasa classify lhandle ehandle ?fgclass?
```

Sub-classify functional groups in the structure according to the LHASA definition. This is used for checking compatibility of standard reaction conditions with functional groups outside the reaction path.

This command is currently not fully implemented.

lhasa collection

```
lhasa collection
Lhasa.Collection()
```

Look into the **Lhasa** transform directory and identify pre-compiled transform files stored therein. The command returns a tuple of the found transform IDs, which are valid identifiers for the **`lhasa read`** command.

lhasa create

```
lhasa create ?ruledata? ?subroutinedata?..
lhasa create -fromfile/-fromfiles rulefilename ?libfilename?...
Lhasa(?ruledata?,?subroutinedata?,...)
Lhasa.Create(?ruledata?,?subroutinedata?,...)
Lhasa.CreateFromFile(rulefilename,?libfilename?,...)
```

Create a new **Lhasa** object and return its object handle. Usually, the content of a transform source file and any required subroutine file contents are specified at creation time. These are directly parsed into an internal byte code representation.

The first command variant expects the rule source code as **`Latin1`** or **`UTF8`** (with **BOM**) string blob data, while the second version expects a set of file names, with the file containing **`Latin1`** or **`UTF8`** (with **BOM**) source code. The first version is useful in contexts where, for example, the rule source is extracted from a database or other non-file source. The second variant automatically sets the *sourcefiles* attribute, which is not possible for the first version because the file name is not directly known.

The rule data can either be the source code for a single transform (with one set of 1D and optionally 2D and New1D **PATRAN** patterns), or a concatenation of one or more classification rules (for example, for the determination of subclasses of functional groups). It is not possible to load more than one transform into a **Lhasa** object, but any number of these objects, with the same or different rule sources, may be instantiated at any time.

If subroutine source code is used, every subroutine source block must be passed as its own argument. It is neither possible to concatenate transform sources and subroutine sources, nor to used merged subroutine source files. Empty subroutine data blobs or filename arguments, or, for filename input, file names which are actually directories are silently skipped. When reading a rule, an attempt is made to automatically load missing library files by trying to locate a file named, in lowercase, like the subroutine name with a *.csr* suffix along the directory path specified in **`::cactvs(lhasapath)`** or its **Python** dictionary equivalent. This only succeeds if the subroutine name to source file mapping follows this scheme - this is the case for most of the original **Lhasa** library files, but not all of them.

Certain object metadata components, such as the transform name and ID, or the substructure match patterns, are automatically extracted from the rule source during parsing and set as object attributes.

The **`lhasa react sub`**command requires the presence of one or more 2D or New1D retro-reaction patterns in the rule source, and these should, on the left - retro-reaction reagent - side, correspond to the 1D patterns as closely as possible, but without excluding any structure motifs which are matched by the somewhat more expressive 1D patterns. Many original **Lhasa** rules to not provide 2D patterns. If that is the case, they must be manually added before a rule becomes fully usable.

The 2D patterns which can be parsed by the toolkit are an extension of the original **Lhasa** implementation. There is no requirement that the *path* is encoded on the reaction arrow line with the *offpath* components drawn above or below this line. Also, aromatic systems are automatically detected if a full ring with proper VB bond orders is drawn. Diagonal double bonds can be encoded as // or \\ character pairs where one of the characters is one position to the left or right of the normal bond continuation. The 2D patterns become subject to a reaction mapping analysis. For performance reasons, it is advisable to use tagged substituent groups like R1, R2 etc instead of anonymous carbon atoms for large patterns.

Example:

```
set lh [lhasa create [read_file huisgen.src]]
set lh [lhasa create -fromfile huisgen.src]
```

lhasa dataset

```
lhasa dataset lhandle ?filterlist?
l.dataset(?filters=?)
```

Return the dataset handle or reference of the dataset the Lhasa object is a member of. It the object is not member of a dataset, or does not pass all of the optional filters, an empty string or **None** for **PYTHON** is returned.

Putting Lhasa objects into datasets is primarily intended for use with the multi-threading **dataset threadexec** command.

Example:

```
lhasa dataset $lhandle
```

lhasa defined

```
lhasa defined lhandle property
l.defined(property)
```

This command checks whether a property is defined for the Lhasa object. This is explained in more detail in the section about property validity checking. Note that this is *not* a check for the presence of property data! The **lhasa valid** command is used for this purpose.

lhasa delete

```
lhasa delete ?lhandle?...
lhasa delete all
l.delete()
Lhasa.Delete(?lhandle/lref?,...)
Lhasa.Delete("all")
```

Delete specific or all Lhasa objects from the toolkit instance. The return value is the number of deleted Lhasa objects.

lhasa dget

```
lhasa dget lhandle propertylist ?filterset? ?parameterdict?
l.dget(property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

lhasa exists

```
lhasa exists lhandle ?filterlist?
l.exists(?filters=?)
Lhasa.Exists(lref=,?filters=?)
```

Check whether a table handle or reference is valid. The command returns boolean 0 or 1. Optionally, the table may be filtered by a standard filter list, and if it does not pass the filter, it is reported as not valid.

lhasa filter

```
lhasa filter lhandle filterlist
l.filter(filters)
```

Check whether the Lhasa object passes a filter list. The return value is 1 for success and 0 for failure.

lhasa get

```
lhasa get lhandle propertylist ?filterset? ?parameterdict?
lhasa get lhandle attribute
l.get(property=,?filters=?,?parameters=?)
l.get(attribute)
l.property/attribute
l[property/attribute]
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For the use of the optional property parameter list and filter arguments, refer to the documentation of the **`ens get`** command.

In addition to retrieving property data, this command is also used to retrieve a large set of attribute values from the Lhasa object. All attributes which can be set (see **`lhasa set`** command) can also be queried. In addition, there are a number of attributes which cannot be set by script commands but are automatically updated when a rule is compiled, or a **`lhasa react`** or **`lhasa score`** command is executed. These are:

- *1dpatterns*
  A list of the ensemble handles or references of the 1D patterns defined as part of the transform.
- *2dpatterns*
  A list of the reaction handles or references of the 2D patterns defined as part of the transform.
- *baserating*
  The base rating of the transform. If individual rating factors are defined in the rule (**`TYPICAL*YIELD`**, etc.), the base rating is computed from a weighted average of these factors, as per the official **LHASA** formula. This supersedes any value defined by a **`RATING`** statement. If only a **`RATING`** statement is part of the rule, its value is returned. If neither rating factors not a plain rating are specified, the default base rating of 50 is substituted.
- *codetable*
  The command returns the handle or reference of a newly generated table object which contains a human-readable representation of the compiled source. When writing this table as a tab-separated file (or directly as an **EXCEL** file), this is a useful tool to debug compilation issues. The table is not associated with the Lhasa object. It must be managed and discarded independently.

- *conditional*flexibility* (or *conditionalflexibility*)
  The enumerated value of the base rating factor as defined in the rule, or *undefined*.
- *coverage*
  The count of opcodes which were evaluated at least once as result of `lhasa score` commands during the lifetime of the **LHASA** object. This number is less than or equal to the total number of opcodes (*opcodecount* attribute) and the same as the number of set bits in the *coveragevector* attribute.
- *coveragevector*
  A bit vector with the length equal to the number of byte-compiled opcodes indicating if an opcode has been evaluated at least once during the lifetime of the Lhasa object as the result of running `lhasa score` commands. This information is used to help constructing test data sets exercising all relevant aspects of a transform.
- *cost*
  The enumerated value of the base rating factor as defined in the rule, or *undefined*.
- *debugtable*
  The table handle or reference of the integrated table object which stores scoring execution trace data if the *debugtable* flag is set in the *execflags* attribute. The table only exists if a trace was executed, and it is re-used for further traced scoring executions. The table is an integral part of the **LHASA** object and is fully managed by it. If no table has been set up, and empty string for **TCL** or `None` for **PYTHON** is returned. The table is used in the **CHMTRN** Web tool for debugging and explaining transform execution.
- *delta*
  The reaction-specific delta value added to the Hammet electrophilic substitution reactivity function value when using such Hammet equation. The value is specified via a `DELTA` statement in the source.
- *description*
  Free-form text extracted from the transform rule source header.
- *forwardtests*
  Get the information from the `FORWARD*REACTION*VERIFICATION` block, if one is present in the transform. This is a list, where each list contains in nested fashion a list of the numbers of reagent molecules from the scan verification block starting with number one, and a second list of all expected forward reaction products as ensembles generated from the forward reaction of the indicated reagents.
- *generalrefs*
  A list of the general (not conditional) literature references defined in the transform.
- *handle*
  The object handle.
- *heteroselectivity*
  The enumerated value of the base rating factor as defined in the rule, or *undefined*.

- *history*
  A nested list/tuple of the authors and dates of transform source code changes, as encoded by **`written*by`** and **`changed*by`** statements. The simpler *author* attribute contains the name of the first author only.
- *homoselectivity*
  The enumerated value of the base rating factor as defined in the rule, or *undefined*.
- *intermol_reagentquery1*
  Return CACTVS query for **`molfile scan`** and variants which checks for the presence of a suitable reagent pattern of the first reagent. The exact match mode depends on the setting of the *countedreagentquery* and *exclusivereagentquery* flags in the *execflags* parameter.
- *intermol_reagentquery2*
  Return CACTVS query for **`molfile scan`** and variants which checks for the presence of a suitable reagent pattern of the second reagent.The exact match mode depends on the setting of the *countedreagentquery* and *exclusivereagentquery* flags in the *execflags* parameter. If no second reagent is defined, the returned query never matches.
- *intermol_reagentquery3*
  Return CACTVS query for **`molfile scan`** and variants which checks for the presence of a suitable reagent pattern of the third reagent. The exact match mode depends on the setting of the *countedreagentquery* and *exclusivereagentquery* flags in the *execflags* parameter. If no third reagent is defined, the returned query never matches.
- *intermol_reagentquery4*
  Return CACTVS query for **`molfile scan`** and variants which checks for the presence of a suitable reagent pattern of the fourth reagent. The exact match mode depends on the setting of the *countedreagentquery* and *exclusivereagentquery* flags in the *execflags* parameter. If no fourth reagent is defined, the returned query never matches.
- *intramol_reagentquery*
  Return a CACTVS query for **`molfile scan`** and variants which checks for the presence of a suitable reagent pattern within a single record. Any ensemble which simultaneously contains a pattern for any variant of the first, second ... up to the number of reagents in a reaction is a match.
- *keywords*
  A free-form list of keywords associated with the transform.
- *killline*
  The transform source code line of the statement which terminated the last rule execution, or the special string values *scoretoolow* and *wrongfragmentcount* if the execution was stopped outside the compiled transform code due to these conditions. The value is 0 if the run was not terminated. It is automatically reset before processing any pattern match in a **`lhasa score`** command.
- *max2dfragments*
  The maximum number of fragments on the product side of 2D or New1D patterns defined in the transform source.

- *mechblock*
  A list of the opcode indices of the first and last **CHMTRN** statements defining the mechanism block.
- *min2dfragments*
  The minimum number of fragments on the product side of 2D or New1D patterns defined in the transform source.
- *new1dpatterns*
  A list of the reaction handles of all New1D patterns defined as part of the transform.
- *newbaserating*
  The computed transform base rating as defined by the rating factors (see *baserating*). If the rule does not define rating factors, an empty string is reported.
- *opcodecount*
  The number of compiled opcodes.
- *opcodes*
  A sequential list of all opcodes byte-compiled from the transform text. Each list element is a dictionary with all attributes applicable to the specific opcode.
- *optionalrefs*
  A list of all optional (conditional) literature references defined in the transform source.
- *orientational*selectivity* (or *orientationalselectivity*)
  The enumerated value of the base rating factor as defined in the rule, or *undefined*.
- *productpatterns*
  A list of all ensemble handles for the product patterns defined by 2D or New1D patterns in the transform. Note that in the original rule text these are written on the left side of the reaction scheme because transforms work in retro-synthetic direction!
- *reactiondataset*
  The handle of the dataset object embedded in the **LHASA** object which contains reaction products after running a `lhasa react` command.
- *reactionproducts*
  A list of all product ensembles generated by the last `lhasa react` command.
- *reactions*
  A list of the reaction handles of all 2D and New1D patterns defined as part of the transform.
- *reagentcount*
  The maximum number of reagents defined in in any 2D or New1D pattern of the transform. This defines also the last supported reagent query (*intermol_reagentquery[1234]*) for a given transform.
- *reagentpatterns*
  A list of all ensemble handles for the reagent patterns defined by 2D or New1D patterns in the transform. Note that in the original rule text these are written on the right side of the reaction scheme because transforms work in retro-synthetic direction!

- *reliability*
  The enumerated value of the base rating factor as defined in the rule, or *undefined*.
- *reputation*
  The enumerated value of the base rating factor as defined in the rule, or *undefined*.
- *rho*
  The reaction-specific multiplier value applied to the Hammet electrophilic substitution reactivity function when using such Hammet equation. The value is specified via a **`RHO`** statement in the source.
- *ruleflags*
  An enumerated list of all the descriptive rule flags set in the transform source code, or *none*. The recognized set is *subgoals*allowed*, *subgoals*not*allowed*, *fga*allowed*, *fga*not*allowed*, *unmasking*, *symmetrical*, *reconnective*, *simplifying*, *disconnective* and *student*. None of these currently has an effect on the execution of the rule code.
- *safety*
  The enumerated value of the base rating factor as defined in the rule, or *undefined*.
- *scantests*
  Get the data from a **`REAGENT*SCAN*VERIFICATION`** blob, if it is present in the transform. The result is a list of ensembles. In property `E_REAGENT_SCAN_VERIFICATION` these ensebmels contain information about their expected reagent roles (intramolecular reagent, intermolecular reagent 1, etc.).
- *scoredataset*
  The handle of the dataset embedded in the **LHASA** object which contains scored reactions after running the **`lhasa score`** command.
- *scoreduplicatecount*
  The number of score pattern matches in the last **`lhasa score`** command which executed without rejection by **`KILL`** or **`DISCONTINUE`** statements in the transform evaluation code, but which were then removed from the scored result set because they were a structural duplicate of an earlier result.
- *scorekillcount*
  The number of score pattern matches which were *killed* during the execution of the transform evaluation code by the last **`lhasa score`** command. The count includes implicit kills by scores decreasing below the kill threshold and similar events.
- *scorematchcount*
  The number of topologically distinct score pattern matches found by the last **`lhasa score`** command.
- *scoredreactions*
  A list of the scored reactions generated as result of the last **`lhasa score`** command.

- *scoretests*
  Get the contents of the **`RETRO*SCORING*VERIFICATION`** block as a list, if such a block is defined in the transform. Each list element consists of a list of the expected outcome (*killed* or *scored*) and the test ensemble handle or reference. If the expected result is a kill, the next list element contains the opcode index (zero-based) of the statement which caused the kill. If the expected result is a successful score, the next element is the final score, followed by another nested list, where each element consists of the zero-based opcode index which produced a score modification (**`ADD`** or **`SUBTRACT`**), followed by the score delta.

- *standardreagentsdataset*
  Get the handle or reference of the dataset object which holds the simple standard reagents collection for Lhasa operations. This is a collection of about 85 bulk chemicals which are used in some Lhasa transforms, but are not usually contained in starting material catalogs. This includes reagents such as water, chlorine gas, ozone, or methanol. The dataset is instantiated and filled with the default set when the first Lhasa engine object is created. It is shared between all Lhasa objects. It can be manipulated by adding, removing or editing the contained structures. If a Lhasa forward reaction reagent set fails to explicitly include any of those small molecules when they are required for the reaction, they are automatically added. See also the **`lhasa react`** command.

- *superatomattachment*
  A boolean value indicating whether the current transform may create leaving groups with superatom components, such as generic alkyl groups. This is extracted from the specifics of the bytecodes of the compiled transform. Not all transform executions with the potential to generate such compounds will do so under all circumstances due to the possibility of encoding multiple reaction paths in the transform.

- *thermodynamics*
  The enumerated value of the base rating factor as defined in the rule, or *undefined*.

- *typical*yield* (or *typicalyield*)
  The enumerated value of the base rating factor as defined in the rule, or *undefined*.

Variants of the **`lhasa get`** command are **`lhasa new, lhasa dget, lhasa jget, lhasa jnew, lhasa jshow, lhasa nget, lhasa show, lhasa sqldget, lhasa sqlget, lhasa sqlnew,`** and **`lhasa sqlshow`**. These commands only work on property data and cannot be used to access engine attributes.

lhasa getparam

```
lhasa getparam lhandle property ?key? ?default?
l.getparam(property=,?key=?,?default=?)
```

Retrieve a named computation parameter from valid property data. If the key is not present in the parameter list, an empty string is returned (**`None`** for **PYTHON**). If the default argument is supplied, that value is returned in case the key is not found.

If the key parameter is omitted, a complete set of the parameters used for computation of the property value is returned in dictionary format.

This command does not attempt to compute property data. If the specified property is not present, an error results.

lhasa jget

`lhasa jget` *`lhandle propertylist ?filterset? ?parameterdict?`*
`l.jget(property=,?filters=?,?parameters=?)`

This is a variant of **`lhasa get`** which returns the result data as a **JSON** formatted string instead of **TCL** or **PYTHON** interpreter objects. The command is usable only for property data, not attribute retrieval.

lhasa jnew

`lhasa jnew` *`lhandle propertylist ?filterset? ?parameterdict?`*
`l.jnew(property=,?filters=?,?parameters=?)`

This is a variant of **`lhasa new`** which returns the result data as a **JSON** formatted string instead of **TCL** or **PYTHON** interpreter objects.

lhasa jshow

`lhasa jshow` *`lhandle propertylist ?filterset? ?parameterdict?`*
`l.jshow(property=,?filters=?,?parameters=?)`

This is a variant of **`lhasa show`** which returns the result data as a **JSON** formatted string instead of **TCL** or **PYTHON** interpreter objects.

lhasa list

`lhasa list` *`?pattern?`*
`Lhasa.List(?pattern=?)`

Return a list of all currently existing Lhasa object handles or references. If a filter pattern is specified, only objects matching it are listed. The pattern syntax is the same as in the standard **TCL** command **`string match`**.

lhasa metadata

`lhasa metadata` *`lhandle property ?field ?value??`*
`l.metadata(property=,?field=?,?value=?)`

Obtain property metadata information, or set it. The handling of property metadata is explained in more detail in its own introductory section. The related commands **`lhasa setparam`** and **`lhasa getparam`** can be used for convenient manipulation of specific keys in the computation parameter field. Metadata can only be read from or set on valid property data.

Valid field names are *bounds*, *comment*, *info*, *flags*, *parameters* and *unit*.

lhasa move

`lhasa move` *`lhandle ?datasethandle? ?position?`*
`l.move(?target=?,?position=?)`

Make the Lhasa object a member of a dataset, or remove it from a dataset. If the dataset handle or reference parameter is omitted, or is an empty string, or **`None`** for **PYTHON**, the object is removed from its current dataset.

If a target dataset handle or reference is specified, the object is added to the dataset, if allowed by the acceptance bits of the dataset, and removed from any dataset it was member of before the execution of the command. By default the object is added to the end of the dataset object list, but the final optional parameter allows the specification of a dataset object list index. The first position is index zero. If the parameter value *end* is used, or the index is bigger than the current number of dataset objects minus one, the object is appended as per the default. It is legal to use this command for moving objects within the same dataset.

Another special position value is *random* or *rnd*. This value moves to the object to a random position in the dataset.

The dataset handle cannot be a transient dataset.

The return value of the command is the dataset of the object prior to the move operation. It is either a dataset handle/reference, or an empty string (**Tcl**) or **None** (**Python**) if it was not member of a dataset.

lhasa mutex

```
lhasa mutex lhandle mode
l.mutex(mode)
```

Manipulate the object mutex. During the execution of a script command, the mutex of the major object(s) associated with the command are automatically locked and unlocked, so that the operation of the command is thread-safe. This applies to builds that support multi-threading, either by allowing multiple parallel script interpreters in separate threads or by supporting helper threads for the acceleration of command execution or background information processing. This command locks major objects for a period of time that exceeds a single command. A lock on the object can only be released from the same interpreter thread that set the lock. Any other threaded interpreters, or auxiliary threads, block until a mutex release command has been executed when accessing a locked command object. This command supports the following modes:

- *lock*
  Increase the recursive mutex lock count on the object. The command returns the current lock count after the command, excluding the transient single-command lock.
- *reset*
  Release all persistent locks on the object, if they exist.
- *test*
  Return the current persistent lock count on the object. This excludes the transient per-command lock.
- *unlock*
  Decrease the recursive lock count on the object. The command returns the current lock count after the command, excluding the transient single-command lock. Unlocking an object which has not been persistently locked results in an error.

There is no *trylock* command variant because the command already needs to be able to acquire a transient object mutex lock for its execution.

The command returns the current lock count.

lhasa need

```
lhasa need lhandle propertylist ?mode? ?parameterdict?
l.need(property=,?mode=?,?parameters=?)
```

Standard command for the computation of property data, without immediate retrieval of results. This command is explained in more detail in the section about retrieving property data.

The return value is the original table handle or reference.

lhasa new

```
lhasa new lhandle propertylist ?filterset? ?parameterdict?
l.new(property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **`lhasa get`** command. The difference between **`lhasa get`** and **`lhasa new`** is that the latter forces the re-computation of the property data, regardless whether it is present and valid, or not.

lhasa nget

```
lhasa nget lhandle propertylist ?filterset? ?parameterdict?
l.nget(property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **`lhasa get`** command. The difference between **`lhasa get`** and **`lhasa nget`** is that the latter always returns numeric data, even if symbolic names for the values are available.

lhasa nnew

```
lhasa nnew lhandle propertylist ?filterset? ?parameterdict?
l.nnew(property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading object data and attributes. It is explained in more detail in the section about retrieving property data.

For examples, see the **`lhasa get`** command. The difference between **`lhasa get`** and **`lhasa nnew`** is that the latter always returns numeric data, even if symbolic names for the values are available, and that property data re-computation is enforced.

lhasa properties

```
lhasa properties lhandle ?pattern? ?noempty?
l.properties(?pattern=?,?noempty=?)
```

Return a list of the valid properties on the Lhasa object. If desired, the property list can be filtered by the optional string match pattern. Since Lhasa objects incorporate no minor objects, only true Lhasa object properties (standard prefix *L_*) are listed.

If the *noempty* flag is set, only properties where at least one data element is not the property default value are output. By default, the filter pattern is an empty string, and the *noempty* flag is not set.

The command may also be written as short form **`lhasa props`**.

lhasa purge

```
lhasa purge lhandle propertylist/table ?emptyonly?
```

```
l.purge(?properties=?,?emptyonly=?)
```

Delete property data from the Lhasa object. In contrast to performing property deletions on, for example, ensembles this operation does not branch out to properties which are stored on objects embedded in the Lhasa objects.

This command only deletes proper Lhasa object properties (usually starting with *L_*). If the object class name *lhasa* is used instead of a property name, the data of all Lhasa object properties is deleted.

The optional boolean flag *emptyonly* restricts the deletion to those properties where the value of a property is identical to the default.

lhasa react

```
lhasa react lhandle ehandle/ensspec ?ehandle/ensspec?...
l.react(ehandle/eref/espec,?ehandle/eref/espec?,...)
```

Generate a comprehensive set of forward reaction products of starting materials according to all matching 2D or New1D patterns from the current transform. If New1D patterns are present, only these are used for matching. The starting materials need to match the right side of the patterns, and the number of specified starting material ensembles must be the same as the number of fragments in the pattern. Every substructure pattern fragment must match on a different starting material molecule from the merged input ensemble in the default *intermolecular* reaction mode, or be part of the same molecule (*intramolecular* mode, see *forwardmode* attribute of the **Lhasa** object). The order of the input structures vs. the left/right sequence of patterns in the transform code is arbitrary. All possible, but symmetry-filtered pattern match variants of all eligible patterns with respect to the input structure set are evaluated.

The starting material structures can either be normal persistent ensemble handles/references or a decodable structure specification (see `ens create`). In the latter case, the decoded input structures are transient and are automatically deleted after the command completes.

The result is a list of all possible duplicate-filtered reaction product ensemble handles or references which could be generated by checking all applicable patterns in all possible match orientations and, upon any successful match, performing the atom and bond changes encoded in the pattern, going from the right side of the pattern to the left. Duplicate result structure filtering is performed according to the configured hash code (see *hashproperty* **Lhasa** object attribute). The reaction products reside in a dataset object embedded in the **Lhasa** object (see *reactiondataset* attribute) and may be further manipulated there, or moved out by additional script commands. The reagent structures, if they are not transient objects, are not consumed.

When this command is run, any existing product ensembles in the reaction dataset are deleted before the execution commences.

If there is no match, this is not an error. The result structure set is simply empty in that case.

The reagent structures used with this command are usually structures found by scanning a starting material catalog with automatically extracted reagent queries (see *reagentquery[1234]* attribute).

A successfully performed reaction does not guarantee that the reverse reaction scoring arrives at the same starting materials. There can be alternative retro-synthetic pattern matches, additional potential starting materials like Cl/Br/I alternative leaving group substitutions, starting materials with generic superatoms like „*alkyl*“ instead of a specific group, markers for required functional group protection, etc. The products generated by this command are only to be used as starting points for retro-synthetic evaluation. The reagent structures returned by **`lhasa score`** need to be matched again vs. a starting material catalog (potentially after attaching protective groups where required, etc.), and the matches from that operation are the real starting materials to be stored in a **LHASA**-scored synthetically accessible structure catalog, not the input structures used in this command.

lhasa read

```
lhasa read bytecodefilename/trafoid
Lhasa.Read(bytecodefilename/trafoid)
```

Create a **LHASA** object by reading a compiled byte code file (see **`lhasa write`** command). The argument is either a single file name or transform ID. Any library subroutine files which are required and were specified when the transform was compiled from source (see **`lhasa create`**) are part of the single byte code file.

It is possible to specify input pipes (with a leading vertical bar in the name) and *stdin* as input. If the file is not specified as full path name, and not readable in the current directory, an attempt is made to find it in the standard transform directory. If the filename is missing the standard *.clb* suffix, it is automatically appended. If the file name is a simple integer transform ID, a short file name is constructed from this identifier and searched. The default transform collection contains links from the transform ID short file name to the corresponding fully named transform file with the highest version number.

The return value is a new **LHASA** object handle or reference.

lhasa ref

```
Lhasa.Ref(identifier)
```

PYTHON-only method to get a reference of the **LHASA** object from its handle.

lhasa rename

```
lhasa rename lhandle srcproperty dstproperty
l.rename(srcproperty=,dstproperty=)
```

This is a variant of the **`lhasa assign`** command. Please refer the command description in that paragraph.

lhasa reset

```
lhasa reset lhandle
l.reset()
```

Reset the Lhasa object from an empty state. All result structures and reaction currently associated with the object are deleted, and the transform definition is purged.

The command is not used in normal processing. Both **`lhasa react`** and **`lhasa score`** automatically clear their respective result sets when the command is started and can thus be invoked any number of time without resetting.

The command returns the original object handle or reference.

lhasa score

```
lhasa score lhandle ehandle/ensspec
l.score(ehandle/eref/ensspec)
```

This command performs a full retro-synthetic evaluation of a product structure according to the **CHMTRN** script commands encoded in the rule source.

Usually the input structure is one from the result set generated by **lhasa react** from potential starting materials, but this is not required. The examined structure may either be a normal persistent ensemble handle, or a decodable structure specification (see **ens create**). In the latter case, the structure is decoded transiently and automatically deleted when the command finishes.

The command exclusively uses the 1D reaction path specifications encoded in the transform code. All possible, symmetry-filtered matches are found and processed. The result is a duplicate-filtered set of forward reactions, which contain newly created potential reagents and a duplicate of the target structure. Sometimes, transform rules may also encode intermediates. These also become part of the reaction with a suitable E_REACTION_ROLE property value, usually *intermediate*. These result reactions reside in the dataset embedded in the **Lhasa** object (see *scoredataset* attribute) and may be further processed there. Existing score result reactions associated with this **Lhasa** object are deleted whenever the command it run, so they have to be saved or moved out of their dataset before the command is re-used if they are still needed.

Depending on the transform rule coding, the proposed reagents may contain expandable superatoms (*alkyl*, *halogen*) and/or other markers (such as functional groups requiring protection). A second-pass match operation against a starting material catalog is required to obtain real starting materials, and there may be multiple catalog matches if the starting material structure contains variable groups, or different pre-attached protecting groups may be suitable.

A scored structure may be matched by multiple 1D patterns, and any of these may be positioned in multiple different match orientations. Furthermore, multiple reaction variants may be generated from within a transform rule (for example, by means of a **BRANCH** **CHMTRN** statement). Therefore, frequently more than a single reaction is the result of a scoring operation on a single structure. It is also possible that no result reactions are reported for a given structure/transform combination, either because no pattern matches, or all possible reaction paths were discarded by means of **CHMTRN** **KILL** statements, or unmet score thresholds. This is even possible when the tested structure was generated by a **lhasa react** command with the same transform - that command performs only pattern matching and atom/bond updates, not reaction knowledge evaluation.

The return value of the command is the highest score among all found retro-reaction paths for the tested product structure, or 0 if there were no result reactions. Every retro-reaction result object holds detailed transform result and logging information in property X_LHASA_SCORE, including its individual score. The result reactions additionally have atom mapping information in A_MAPPING and may contain additional information on the role of specific atoms in property A_FLAGS (especially in bits *protected*, *interfering*, *participating*).

lhasa scorematch

```
lhasa scorematch lhandle ehandle/espec
l.scorematch(ehandle/eref/ensspec)
```

Test whether a structure intended to be retro-synthetically scored matches the transform patterns for the current transform. No actual scoring is performed. The return value is the total number of distinct matches of the score structure on all retro patterns. If the value is zero, the structure cannot be processed by the current transform. This command helps in distinguishing between unsuitable score structures and those where all reaction paths are killed by the transform code executed after one or more matches.

lhasa set

```
lhasa set lhandle ?property/attribute value?...
lhasa set lhandle ?dictionary?
l.set(property/attribute,value,...)
l.set({property/attribute:value,...})
l.property/attribute = value
l[property/attribute] = value
```

Set one or more attributes of the **LHASA** object. All attributes which can be set can also be queried via the **lhasa get** command. Attributes which cannot be modified are listed under **lhasa get**.

These are the recognized attributes:

- *address_city*
  The city part of the author contact address.
- *address_country*
  The country part of the author contact address, following the ISO3166 standard.
- *address_state*
  The state part of the author contact address. Empty if not applicable.
- *address_street*
  The street address part of the author contact address. Includes floor, house number, etc.
- *address_zip*
  The **ZIP** code or other applicable postal code of the author contact address.
- *affiliation*
  The institution the author of the transform rule works for.
- *affiliationduns*
  The **DUNS** registration ID of the affiliated institution. This is primarily useful for US government projects.
- *affiliationurl*
  The **URL** of the affiliated institution.
- *author*
  The (first) author of the transform rule . This is a free-form string. When a transform is read from source or compiled code, this is set to the value encoded in the first **written*by** or **changed*by** statement.
- *authorurl*
  A **URL** with information on the author, or an empty string if unset.
- *category*
  A category string to be used if the rule definition is stored in a repository.
- *classuuid*
  The base class **UUID** of this rule definition.

- *comment*
  A free-form text comment.

- *complexityfilter*
  A maximum complexity value (as per property `E_COMPLEXITY`) which is acceptable for starting material selection. If the attribute is set to a value larger than zero (the default), a query condition is automatically added to the auto-generated starting material queries (read-only attribute *intermol_reagentquery1* etc., see **`lhasa get`** command). This attribute can be combined with the element filter attribute.

- *date*
  This attribute stores the date of the rule definition compilation. It is used for information purposes only. If a new rule is compiled, it is initialized to the current date. When a transform is read from source code, it is set to the last date encoded in **`written*by`** or **`changed*by`** statements.

- *debugimagesize*
  The height and width of images measured in 72dpi pixels which are entered in the debug table (see *debugtable* and *execflags* attributes) for visualizing the progress of scoring operations. The default image size is 200 pixels.

- *doi*
  A digital object identifier for the rule, if defined.

- *elementfilter*
  An element filter (formula match) expression (see **`molfile scan`** command). If it is specified as a non-empty string, it is added as an additional criterion to the auto-generated starting material queries (read-only attribute *intermol_reagentquery1* etc., see **`lhasa get`** command). The attribute can be combined with the complexity filter attribute.

- *email*
  The email address of the creator of the rule, useful in case a definition file is distributed and anybody has questions.

- *execflags*
  A collection of flags modifying various aspects of the operation of the object. The recognized flags are:

  - *checkreactionbonds* - if set, the breaking or creation of bonds in retro-synthetic (scoring) direction only succeeds if the bonds are marked in property `B_REACTION_CENTER` as changed, created or broken. This marking of bonds in this property is automatically applied when the forward reaction for retro analysis is created (**`lhasa react`**). This option prevents possible deviations of bond transformations into out-of-pattern regions of the product molecule. However, if the retro-analyzed product does not contain `B_REACTION_CENTER`, for example because the scored product was not generated via the standard mechanism, the analysis fails.

- *checkreagents* - this flag has only an effect if the scored molecule is the product component of a reaction, where the reagents component contains the starting materials. If set, a check is performed if all, of the retro-synthetically generated fragments match molecules in the reagents component. If not, the generated score structure is discarded. This flag is usually combined with the *checkreactionbonds* flag and further restricts the score result set. For example, an asymmetric ether usually generated (in absence of interfering features) 6 different score solutions with the Williamson ether synthesis transform - attaching a halogen on the fragment on both sides of the ether oxygen, and proposing the use of chlorine, bromine and iodine as reacting halogen. With the *checkreactionbonds* flag and properly set properties on the scored structure, only the side of the ether which was reacted in the forward reaction is cut since this encoded in bond property `B_REACTION_CENTER`. However, this still generates three solutions since the scored product on its own has no knowledge which halogen was used when it was generated from starting materials. If however the scored structure is part of a full forward reaction, only the halogen variant used in its generation is reported as unique scored result.
- *countedreagentquery* - if set, reagent queries check for single-instance presence of the substructure, discarding structures with multiple pattern matches in non-overlapping locations.
- *debugtable* - Create and fill an embedded table object with a transform execution trace. This is internally used in the **CHMTRN** Web tool.
- *dumperrorstructure* - If a structure triggers an error during retro analysis, dump it as a serialized object into a file in the current directory.
- *exclusivereagentquery* - if set, suitable reagents do not just need to contain the pattern of a reagent specification, but additionally cannot contain the patterns of any other reagent, except for intramolecular reagents.
- *forwardreactiontrace* - Trace the execution of forward reactions.
- *forwardresultsasreactions* - Normally, the forward application of a transform (**`lhasa react`** command) returns the result structures as simple structures, not as part of a full reaction with the starting materials. If this flag is set, the result structures are part of a reaction, with `B_REACTION_CENTER` and `A_MAPPING` set on both reagent and product ensemble. This is the preferred mode to generate suitable scoring structures for use with the *checkreagents* flag.
- *forwardscorabilitycheck* - if set, perform an extra check on whether the scored molecule matches any of the score patterns, without executing any real scoring.
- *forwardstereoexpansion* - if set, forward reactions which generate stereogenic centers and bonds in the course of the reaction do not return undefined stereochemistry at these, but an enumeration of molecules with defined stereochemistry, allowing the selection of stereo-specified starting materials from retrosynthetic analysis.
- *keepkills* - if set, keep killed structures and report them in the score result set with a -999 score, but discontinue execution of the transform.

- *nosuperatoms* - Do not attach leaving groups with superatoms like [ALK] in retro analysis. Instead, attach a specific minimal leaving group (usually a C1 fragment).
- *prefernew1dscorepatterns* - if set, prefer the use of the **NEW1D** pattern(s) not just for forward reactions, but also for scoring, provided they exist. By default, the original **1D** pattern(s), if they exist, have precedence for this purpose. In most cases, the left-side portion of a **NEW1D** pattern is identical to the equivalent **1D** pattern, except in cases where the reaction encoding of the **NEW1D** pattern needs to avoid generic bond types.
- *reagentequivalencytrace* - A trace option for determining the equivalency of reagents generated during retro analysis.
- *registerscore* - Register score events and circumstances for later detailed analysis.
- *reagentmatchtrace* - A trace option for reagent matching.
- *reversepatternmatchtrace* - A trace option for the pattern matching in retrosynthetic analysis.
- *symmetricreagentquery* - If more than a single non-overlapping match of the patterns of potential starting materials is allowed (see *targetmatchcount* attribute), multiple matches are only allowed if they are topologically equivalent. In the default case of allowing only single matches, this flag has no effect.

The default execution flag value is *exclusivereagentquery|countedreagentquery.*

- *forwardmode*
  Change the way forward reactions (**lhasa react**) are processed. Possible values are *intermolecular* (the default) and *intramolecular.* In intermolecular mode, the reagent patterns of a possible reaction must match onto different molecules in the combined reagent set, and the number of molecules in the set must be the same as the number of disconnected patterns in the matched reaction (usually 2, but up to 4, depending on how the **2D** or **New*1D** pattern is encoded). In intramolecular mode, there can be only one reagent molecule, and all reagent patterns must match in non-overlapping fashion the same molecule.
- *hashproperty*
  The name of the hash code property used to eliminate structure duplicates. The default is `E_HASHISY`. New 128-bit hash codes are not yet supported.
- *id*
  The official registration ID of the transform rule. *regid* is an alias.
- *interferingcolor*
  The color to use to mark interfering functional groups in transform result graphical reports. The default is „light salmon“.
- *infourl*
  A **URL** with information on the transform rule, or an empty string if unset..
- *license*
  The license class associated with this transform rule. Setting the license to a standard type updates the associated **URL** with a standard location.

- *licenseurl*
  A **URL** with details about the rule license.
- *maxqueryfragments*
  The maximum number of molecules in a matching reagent record. The default value is 1, i.e. allow only reagents without counterions, etc.
- *mininorganicsize*
  The minimum heavy atom count of inorganic fragments generated in retro scoring mechanisms to retain the fragment in the result ensemble. The default is 2, i.e. inorganic fragments such as water with one heavy hetero atom are deleted.
- *minmolsize*
  The minimum heavy atom count a fragment must meet to be not automatically discarded when a result structure ensemble is logged. The default is 2.
- *name*
  This is the name of the transform rule. Changing it after it has been defined originally is generally considered bad style, but it is possible.
- *orcid*
  The **ORCID** code of the author of the transform rule (see www.orcid.org).
- *participatingcolor*
  The color used for marking reaction-participating (in **CHMTRN** parlance) functional group in transform result graphical reports. The default is *palegreen*.
- *phone*
  A contact phone number of the author.
- *protectedcolor*
  The color used for marking functional groups which need protection under the conditions of the retro-reaction. The default is *plum*.
- *reactioncycles*
  The number of iterative cycles the `lhasa react` command should perform when it is run. The default is one, and any other value is useful only under special circumstances.
- *reagentexclusionmode*
  The mechanism used to exclude parts of potential reagents from matching the reagent pattern because they are already matched by patterns of reagents in other roles. Useful values are combinations of *burnatoms*, *burnbonds*, *burncarbon*, *burnterminals*, *burnringsystems*, and *burnaroringsystems*. The default is *burnatoms|burnaroringsystems*. Please refer to the `match` or `molfile scan` commands for additional information.
- *reagentmatchmode*
  The substructure match mode to use to match starting materials. This is a standard substructure match parameter and possible value are explained in the section on the `match` command. The default mode is *conditionalnocommonfgatoms*.
- *references*
  Cross references of the rule. This is a nested list of class **UUID**s and reference type tags.

- *scorethreshold*
  A positive integer value which specifies a threshold for the accumulated score. If, at the time a result structure is to be reported, its score is below the threshold, reporting is skipped, even though the result structure was not explicitly **`KILL`**ed. If the attribute is set to a negative value (which is the default), no result structure score filtering is performed.

- *scoredmolcount*
  An integer value which can be set to indicate the required number of molecular fragments in the result reagent after executing a **`lhasa score`** command. If the number is not negative, only results which match the number are accepted - other outcomes are **`KILL`**ed. This is useful to exclude (set to 2 or other number of reagent structures) or require (set to 1) intramolecular retro-reactions. If set to a negative value (the default), not result fragment count filtering is performed.

- *sourcefiles*
  A list of the names of the files which contained the source code for the compiled transform rule. If the rule was compiled from blob data without reading named files (see **`lhasa create`** command) all names are set to an empty string. If this attribute is set, the number of list elements must be the same as the number of data blobs or source files used to compile the rule.

- *stereosynthesismode*
  This attribute can be set to *none* (the default), *eantioselective* or *diastereoselective*. The set value is checked against **CHMTRN** target keywords *ENANTIOSELCTIVE*SYNTHESIS* and *DIASTEREOSELECTIVE*SYNTHESIS* if they are encountered in transform rules while executing the **`lhasa score`** command.

- *subtractthreshold*
  A positive integer value which leads to the re-interpretation of a **`SUBTRACT`** **CHMTRN** statement as a **`KILL`** statement if the final negative delta is larger or equal to the specified threshold. The value only applies to the delta of a single **`SUBTRACT`** statement, not the accumulated score. If set to a negative value (which is the default), no command re-interpretation is performed regardless of the demerit.

- *targetmatchcount*
  Set the desired number of non-overlapping reagent pattern matches in potential starting materials. This number is 1 by default.

- *traces*
  A list of conditions which limit the range of transform execution traces activated via setting the **`cactvs(trace)`** control variable to *lhasa*. By default, all executed byte-code statements are traced, but trace output can be limited to code sections of interest. The argument is a list of trace sections. Every list element is a nested list of one to three integers. The first element of the nested lists is the start line in the source file of the traced section. Code before that line is not traced. The optional second element is the last source line of the trace section. If it is not set, the trace extends to the end of the source file. The optional third element is the source file index, with zero for the base rule file and one or higher for additional library files. The default value for this element is zero, i.e. if a trace section is specified without an explicit source file index, it applies to the base transform file.

- *version*
  The version of the node. This is a string in a 1.2.3 (or shortened) style.
- *versionuuid*
  The version **UUID** associated with this transform rule version.

lhasa setparam

```
lhasa setparam lhandle property ?keyword value?...
lhasa setparam lhandle property dictionary
l.setparam(property,?key,value?...)
l.setparam(property,dict)
```

Set parameter values in the metadata section of existing property data attached to the Lhasa object. This command does not change the parameters for computations in the property definition (see **prop setparam** command for this function). It only stores its data in the parameter set which was copied into the metadata when the property was computed for the table.

This command does not attempt to compute property data. If the specified property is not present, an error results.

lhasa show

```
lhasa show lhandle propertylist ?filterset? ?parameterdict?
l.show(property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **lhasa get** command. The difference between **lhasa get** and **lhasa show** is that the latter does not attempt computation of property data, but raises an error if the data is not present and valid. For data already present, **lhasa get** and **lhasa show** are equivalent.

lhasa sqldget

```
lhasa sqldget lhandle propertylist ?filterset? ?parameterdict?
l.sqldget(property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **lhasa get** command. The differences between **lhasa get** and **lhasa sqldget** are that the latter does not attempt computation of property data, but initializes the property value to the default and returns that default, if the data is not present and valid; and that the **SQL** command variant formats the data as **SQL** values rather than for **TCL** or **PYTHON** script processing.

lhasa sqlget

```
lhasa sqlget lhandle propertylist ?filterset? ?parameterdict?
l.sqlget(property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **`lhasa get`** command. The difference between **`lhasa get`** and **`lhasa sqlget`** is that the **SQL** command variant formats the data as **SQL** values rather than for **TCL** or **PYTHON** script processing.

lhasa sqlnew

```
lhasa sqlnew lhandle propertylist ?filterset? ?parameterdict?
l.sqlnew(property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **`lhasa get`** command. The differences between **`lhasa get`** and **`lhasa sqlnew`** are that the latter forces re-computation of the property data, and that the **SQL** command variant formats the data as **SQL** values rather than for **TCL** or **PYTHON** script processing.

lhasa sqlshow

```
lhasa sqlshow lhandle propertylist ?filterset? ?parameterlist?
l.sqlshow(property=,?filters=?,?parameters=?)
```

Standard data manipulation command for reading object data. It is explained in more detail in the section about retrieving property data.

For examples, see the **`lhasa get`** command. The differences between **`lhasa get`** and **`lhasa sqlshow`** are that the latter does not attempt computation of property data, but raises an error if the data is not present and valid, and that the **SQL** command variant formats the data as **SQL** values rather than for **TCL** or **PYTHON** script processing.

lhasa transfer

```
lhasa transfer lhandle propertylist ?targethandle? ?targetpropertylist?
l.transfer(properties=,?target=?,?targetproperties=?)
```

Copy property data from one Lhasa object to another Lhasa object or other major object, without going through an intermediate scripting language object representation, or dissociate property data from the Lhasa object. If a property in the argument property list is not already valid on the source object, an attempt is made to compute it.

If a target object is specified, the return value is the handle or reference of the target object. The source table and the target object cannot be the same object.

If a target property list is given, the data from the source is stored as content of a different property on the target. For this, the data types of the properties must be compatible, and the object class of the target property that of the target object. No attempt is made to convert data of mismatched types. In case of multiple properties, the source property list and the target property list are stepped through in parallel. If there is no target property list, or it is shorter than the source list, unmatched entries are stored as original property values, and this implies that the object class of the source and target objects are the same.

If no target object is specified, or it is spelled as an empty string or **PYTHON `None`**, the visible effect of the command is the same as a simple **`lhasa get`**, i.e. the result is the property data value or value list. The property data is then deleted from the source object. In case the data type of the deleted property was a major object (i.e. an ensemble, reaction, table, dataset or network), it is only unlinked from the source object, but not destroyed. This means that the object handles or references returned by the command can henceforth the used as independent objects. They can be deleted by a normal object deletion command, and are no longer managed by the source object.

lhasa valid

```
lhasa valid lhandle propertylist
l.valid(property/propertysequence)
```

Returns a list of boolean values indicating whether values for the named properties are currently set for the Lhasa object. No attempt at computation is made.

`lhasa has` is an alias to this command.

lhasa verify

```
lhasa verify lhandle property
l.verify(property)
```

Verify the values of the specified property on the Lhasa object. The property data must be valid, and a table property. If the data can be found, it is checked against all constraints defined for the property, and, if such a function has been defined, is tested with the value verification function of the property.

If all tests are passed, the return value is boolean 1, 0 if the data could be found but fails the tests, and an error condition otherwise.

lhasa write

```
lhasa write lhandle ?filename?
l.write(?filename=?)
```

Write a byte code file encapsulating the currently loaded and compiled transform or classification rule. This file can be used as input for a **`lhasa read`** command to restore the **LHASA** object at a later time. Only the rule content and the object metadata is stored, not the current execution state and result data of the **LHASA** object. The byte code contains the code generated from all included **CHMTRN** library files, not just the primary transform file, if applicable.

If no file name is specified, it is constructed from the **LHASA** object name and the suffix *.clb* (**CACTVS LHASA** Binary) without a directory path component. It is possible to use output pipes or (on Linux/Unix) **TCL** or **PYTHON** file handles or references as special output file names. If an explicit empty filename is set (including **`None`** for **PYTHON**), no disk file is written. Instead, the file contents are returned as byte blob.

The primary purpose of the binary format is distribution of rule content without access to the source code. We have permission from the **LHASA** rights holders to distribute binary versions of original **LHASA** transforms, but not their source code. Transform source code is only available as part of the standard toolkit package for newly added transform rules not authored by Lhasa Inc. or Lhasa Ltd. Additional source may be made available for customers with an extended license or the legal successors of the members of the original industry consortium sponsoring the development of these rules. In addition, a large part of the Lhasa UK source part has been put into the public domain.

The binary format in this **`CHMTRN`** interpreter implementation is intentionally not compatible to the original **`Lhasa`** byte code, avoiding its limitations due to insufficient number of bits in the 32-bit values of the virtual processor code.

The return value of the command is the filename, which may have been auto-generated, except in case when the output is a blob.

The netx Command

The **`netx`** command is used to manage network file I/O modules. The command has the following subcommands:

netx defined

```
netx defined networkformat
Netx.Defined(networkformat)
```

A boolean check whether the network format is supported by a module. If the format is not yet known, an attempt is made to locate and auto-load the module. For an equivalent command without auto-loading, see **`netx exists`**. The name can be the primary name of the format, or any recognized alias.

netx exists

```
netx exists networkformat
n.exists()
Netx.Exists(networkformat)
```

A boolean check whether the network format is supported by a module. No attempt is made to auto-load a handler module if it is not already in memory. The name can be the primary name of the format, or any recognized alias. For an equivalent command with auto-loading, see **`netx defined`**.

netx get

```
netx get networkformat attribute
Netx.Get(networkformat,attribute)
n.get(attribute)
n.attribute
n[attribute]
```

Query the value of an attribute of the module Note that network format handlers are static - it is neither possible to define them on the command line, nor to change any attribute. Therefore, there are no **`netx create`** or **`netx set`** commands.

The following attributes are recognized:

- *address_city*
  The city part of the author contact address.
- *address_country*
  The country part of the author contact address, following the ISO3166 standard.
- *address_state*
  The state part of the author contact address. Empty if not applicable.
- *address_street*
  The street address part of the author contact address. Includes floor, house number, etc.
- *address_zip*
  The **ZIP** code or other applicable postal code of the author contact address.
- *aliases*
  A list of recognized alias names of the format.

- *affiliation*
  The institution the author works for.
- *affiliationduns*
  The **DUNS** registration ID of the affiliated institution. This is primarily useful for US government projects.
- *affiliationurl*
  The **URL** of the affiliated institution.
- *author*
  The author of the module, as free-format text.
- *authorurl*
  A **URL** with information on the author, or an empty string if unset.
- *builtin*
  A boolean flag indicating whether this is a built-in type.
- *capabiltities*
  A collection of flags reporting specific attributes of the module. The currently supported set is:
  - *none*
    No flags.
  - *read*
    Module supports input from a file.
  - *write*
    Module support output to a file.
- *category*
  A category string to be used if the module is stored in a repository.
- *classuuid*
  The base class **UUID** of this module.
- *comment*
  A free-form comment.
- *date*
  The date the module source code was last changed.
- *doi*
  A digital object identifier for the module, if defined.
- *email*
  An email contact address of the developer of the module.
- *features*
  A list of the functions this module provide. Currently, these can be *check* (format auto-detection), *read* (format input) and *write* (format output).
- *infourl*
  A **URL** with information on the module, or an empty string if unset.
- *keywords*
  A list of keywords associated with the module.

- *license*
  The license class associated with this module. Setting the license to a standard type updates the associated **URL** with a standard location.
- *licenseurl*
  A **URL** with details about the module license.
- *literature*
  A free-form literature reference.
- *mimetype*
  The **MIME** type associated with the format.
- *name*
  The primary name of the format. Since the information may be queried via an alias name, this can be different from the command argument.
- *orcid*
  The **ORCID** code of the author (see www.orcid.org).
- *path*
  The repository path for displaying hierarchical repository trees. This attribute is independent of any file system paths.
- *phone*
  A contact phone number of the author.
- *references*
  Cross references of the module. This is a nested list of class **UUID**s and reference type tags.
- *regid*
  For officially registered modules, this is the assigned registration ID. Unregistered modules report zero.
- *slot*
  The slot in the handler table the module was loaded into.
- *suffixes*
  A list of file name suffixes commonly associated with the format.
- *version*
  Version information for the module. This is a string in a 1.2.3 (or shortened) style.
- *versionuuid*
  The **UUID** associated with this module version.

netx list

```
netx list ?pattern?
Netx.List(?pattern=?)
```

Return a list of all currently loaded network file modules, including those handled by built-in modules. If desired, the list can be filtered by a string pattern.

netx ref

```
Netx.Ref(networkformat)
```

PYTHON-only method to get a reference of the module, which allows terser attribute retrieval commands and other operations.

netx subcommands

```
netx subcommands
dir(Netx)
```

Return a list of the currently implemented subcommands of the **`netx`** command.

The prop Command

The **`prop`** or **`property`** command is used to create, modify, query and delete property definitions.

The command has the following subcommands:

prop addparameterdict

```
prop addparameterdict propertyname parametername datadict
Prop.Addparameterdict(property/pref,parametername,dict)
p.addparameterdict(parametername,dict)
```

Add a new parameter to a property definition. The recognized dictionary items are the same as in the **`prop getparamdict`** command, with the exception that all parsed values are ignored - parameter values, defaults, min and max values are defined via string data, for which a parse is attempted and an error generated if it fails. All dictionary items are optional. If a parameter definition attribute is not defined in the dictionary, an empty or undefined default parameter specification field value is entered.

If the named parameter is already defined, an error is produced.

The command returns the property name or handle.

`prop addparamdict` is an alias to this command.

prop alias

```
prop alias
prop alias propertyname
prop alias propertyname aliasname ?propertyname aliasname?...
Prop.Alias()
Prop.Alias(propertyname/pref)
Prop.Alias(propertyname/pref,aliasname,?property/pref,aliasname?...)
Prop.Alias(dict)
p.alias(?aliasname=?)
```

This command defines an alias name for a property name. After defining an alias, any time the alias name is used as a property name, it is silently and automatically translated to the original property name. It is possible to map an alias name to another alias. Name resolution recursively proceeds until the name can no longer be resolved. Aliases have precedence over basic property names. If an alias name is the same as an existing property name, that existing property becomes effectively invisible and is replaced by the property the alias links to.

The property or alias name the new alias maps to must be resolvable at the time of definition. Alias names must conform to the **CACTVS** property naming scheme, e.g. is not not possible to define arbitrary aliases. This is not an equivalent for setting the *originalname* property attribute.

In case the command is used without arguments, it returns a dictionary of current alias definitions. The keys are the recognized alias names and the values the property or second-level alias name the keys are translated to. If only the property name is specified, a list of the aliases for this property is reported.

Example:

```
prop alias E_NATOMS E_ATOM_COUNT
```

After execution, the number of atoms in an ensemble can both be queried via property `E_NATOMS` (original name) and `E_ATOM_COUNT` (new alias name).

The default start-up scripts of the toolkit include a set of alias definitions which map older, less systematic property names as aliases to current official property names.

prop cast

```
prop cast srcproperty/srcdatatype dstproperty/dstdatatype value
p.cast(destination=,value=)
Prop.Cast(srcproperty/srcdatatype,dstproperty/dstdatatype,value)
```

This command is used to convert or reformat property data. Its arguments are different from that of other **`prop`** subcommands. In addition to a simple property name, an indexed property name or the name of a toolkit property data type can be specified both as source and target description.

The *value* argument is parsed into a toolkit data item using the input routines associated with the property datatype, property field datatype, or directly specified datatype. All data format variants supported by a data manipulation command such as **`ens set`** are allowed, and different input formats may lead to the same internal representation.

The data item is then cast to the target datatype, and re-output as **Tcl** or **Python** interface object using the output routines associated with the target data type. There is no permanent storage of the value in any object. If there is a direct cast path encoded between the source and target data types, it is used, otherwise internally an attempt is made to convert the data item to a string and re-parse it with the target data type I/O routines.

The return value of the command is the argument value encoded in the target data type and/or field format.

Examples:

```
prop cast int string 3
prop cast E_NAME E_NAMESET "a single name"
```

The first line casts an integer 3 to a string with characters "3". The second example casts the input value from string (the data type of `E_NAME`) to a single-element string vector (the datatype of `E_NAMESET`).

prop compare

```
prop compare propertyname value1 value2 ?cmpflags? ?auxvalue1? ?auxvalue2?
prop compare propertyname(field) value1 value2 ?cmpflags? ?auxvalue1? ?auxvalue2?
prop compare datatypename value1 value2 ?cmpflags? ?auxvalue1? ?auxvalue2?
p.compare(value1=,value2=,?comparisonflags=?,?auxvalue1=?,?auxvalue2=?)
Prop.Compare(propertyname?(field)?/pref?,value1=,value2=,?comparisonflags=?,
   ?auxvalue1=?,?auxvalue2=?)
Prop.Compare(datatypename/dref,value1=,value2=,?comparisonflags=?,?auxvalue1=?,
   ?auxvalue2=?)
```

Compare two property values. In the first two forms, the name of the property, or of a property field, is used primarily to get the data type for the decoding of the comparison values and the selection of the applicable comparison function. Alternatively, a data type name can also be supplied directly.

The value parameters are expected to be suitable string or interpreter language representations for the underlying data type. If the property name form of the command is used, they may be specified as enumerated values and other formats which can only be decoded with the aid of information attached to a specific property. The comparison values are converted using the string or **Tcl** or **Python** object input function associated with the data type. If the property name form is used, they are also subject to any other tests associated with the property, such as constraints or regular expression pattern matches.

The *cmpflags* argument is a list of words which are used to select specific comparison function modes. The default is an empty list. In that case, no modification flags are passed to the comparison function. In case a mode bit is selected which is not supported by a specific comparison function, it is silently ignored. Some of the flags may be combined to yield specific useful comparison results, but it is the responsibility of the implementation of a data-type specific comparison function to determine which flags can be combined, and which have precedence in case of conflicts. The following words are currently recognized:

- *none*
  Equivalent to an empty string, no bits set
- *absolute*
  For numerical comparisons, use the absolute value.
- *anymatchelement*
  In case of data types with separate elements (vectors, etc.) check whether any element in the first value is identical to the corresponding element in the second vector. If yes, return 1, else 0.
- *anymisselement*
  In case of data types with separate elements (vectors, etc.) check whether any element in the first value is different from the corresponding element in the second vector. If yes, return 1, else 0.
- *approximate*
  Perform an approximate comparison. For string types, this means that white space, punctuation characters, digits and character case are ignored. If combined with the *withdigits* flag, the comparison takes digits into account, but still ignore whitespace, punctuation and case.

  For floating-point comparisons, this flag indicates that rounded integers should be used for comparison, not the float values.

  If the toolkit was compiled with support for the **TRE** library, this attribute can also be used in combination with a regular expression comparison. In that case, the result is a percentage similarity score between the query expression and the test string, with character substitutions, deletions and insertions all having the same weight.
- *asnumber*
  In case of strings, compare string contents as signed floating point numbers instead of characters.
- *bitcount*
  For bit vector and integer types, compare the number of set bits, not the numerical value.

- *bitset*
  For bit vector and integer types, check whether all set bits in the second argument are also set in the first argument. The return value is -1 if the second argument contains set bits which are not in the first argument, 0 if the values are identical, and 1 if the first value contains set bits that are not set in the second argument. In case there are set bits in either of the two arguments with no corresponding set bit in the other argument, the return value is -1.
- *bitunset*
  For bit vector and integer types, check whether all unset bits in the second argument are also unset in the first argument. The return value is -1 if the second argument contains unset bits which are not in the first argument, 0 if the values are identical, and 1 if the first value contains unset bits that are not unset in the second argument. In case there are unset bits in either of the two arguments with no corresponding unset bit in the other argument, the return value is -1.
- *contained*
  Check whether the value of the second argument is contained in the first. The return value is 1 if it is, 0 otherwise. For strings, this is a substring search. For integers and bit types, it counts the number of common bits. For vectors, it checks whether all elements of the second argument are also found in the first vector, but there is no requirement that the matching element indices are the same.
- *correlation*
  For bit types and numerical vector types, compute the correlation coefficient between the vector arguments. The result is multiplied by 100 and rounded to the next integer.
- *cosine*
  This operation is only supported for bit vectors. It returns the cosine similarity coefficient as an integer, multiplied by one hundred.
- *dice*
  Compute scaled (0..100) Dice similarity coefficient for bit vectors, bit sets and strings (via bigraphs).
- *dictionary*
  Compare strings in dictionary order.
- *euclid*
  For bit types and numerical vector types, compute the Euclidean distance between the points represented by the vector arguments. The result is rounded to the next integer.
- *extended*
  For string comparisons with regular expressions, use extended regular expression syntax.
- *glob*
  For string comparisons, the second value is interpreted as a shell-style glob expression. The return value is 1 if the expression matches, 0 otherwise. The position of the expression argument vs. the simple string can be changed with the *swap* flag.
- *ignorecase*
  For string-type comparisons, ignore character case.

- *ignoredashes*
  Ignore any dash/minus characters in string comparisons. This is useful for comparing **CAS** numbers with non-standard separator locations.
- *ignorewhitespace*
  For string comparisons, ignore white space, but not character case.
- *left*
  For string comparisons, match the left side of the target strings.
- *like*
  For string comparisons, interpret the comparison value as **SQL**-style *like* pattern.
- *precision*
  use the precision as defined by the property attribute *precision* for comparison, instead of the native data type precision. If the first optional parameter of the command is specified, its value is used instead of the precision set in the property definition. The precision value an integer, defining the number of significant digits after the decimal point. Negative values are allowed to define significance limited to differences in tens or hundreds in a property value.
- *overlap*
  This operation is only supported for the float pair type. For each value, the pair defines a minimum and maximum range. If the ranges overlap, zero is returned, otherwise -1 (in case the first range is entirely to the left) or 1 (in the opposite case).
- *regexp*
  For string comparisons, the second value is interpreter as a regular expression. The return value is 1 if the expression matches, 0 otherwise. This flag can be combined with the *ignorecase* and *extended* flags, which further modify the interpretation of the regular expression. The argument position of the expression argument vs. the comparison string can be changed with the *swap* flag. Starting with toolkit version 3.352, the regular expression syntax on all platforms is that of the **PCRE** library, i.e. the **PERL** style.
- *right*
  For string comparisons, match the right part of the target string.
- *swap*
  Exchange the left/right roles of the arguments. This also affects the special interpretation of arguments in certain positions, as with the *glob* and *regexp* modifiers.
- *trim*
  In case of string comparisons, ignore leading and trailing white space, but not embedded.
- *tanimoto*
  For bit types and numerical vector types, compute the Tanimoto coefficient. The result is multiplied by 100 and rounded to the next integer.

- *tversky*
  For bit vectors, compute the Tversky score between the vector arguments. This is one of the few modes where the extra arguments have an effect. Here, they are expected to be values in the 0..100 range. They are divided by 100 and used as floating-point *c1* and *c2* parameters. In case both are set to 50, the result is the same as a Tanimoto comparison. Note that the default values of zero for the extra arguments are not useful for this mode and reasonable values must be supplied.
- *withdigits*
  For approximate string comparisons, recognize and compare digits. By default, they are ignored in that mode. In any case, approximate string comparisons ignore white space and punctuation.

The return value is the integer value returned by the comparison function. For most modes, this is either 0 (values are equal), 1 (left value is larger) or -1 (right value is larger). Some modes do however return other values. For example, similarity bit vector or bit set comparisons return a value between 0 and 100.

The default values for the optional extra parameters are both zero.

Example:

```
prop compare E_SCREEN [ens get $eh1 E_SCREEN] [ens get $eh2 E_SCREEN] tanimoto
```

This command performs a Tanimoto comparisons on the screening vectors of the two ensembles and returns a score in the range 0..100.

prop configure

```
prop configure propertyname
```

Execute the property configuration function (attribute *configfunction*). If there is no function defined, or the execution of the configuration function fails, the command does nothing. By convention, a property configuration function opens a **Tk** window with various GUI elements for the convenient adjustment of computation parameters. Because this is usually a graphical operation, and the overhead of extending the property slave interpreter to a fully GUI-enabled version, this command is always executed in the global interpreter.

Note that this function usually does not directly result in the change of property attributes and parameters. The standard approach is to set up and display a **Tk** window, which remains open until it is closed by the user, while this command returns immediately. Changes from adjustments in the property panel take effect asynchronously only after an *apply* button has been pressed by the user.

This function is not supported in the **Python** interface.

prop create

```
prop create propertyname ?attribute value?...
prop create propertyname dict
Prop(propertyname,?attribute,value?,...)
Prop(propertyname,dict)
Prop(propertyname,?attribute=value?,...)
Prop.Create(propertyname,?attribute,value?,...)
Prop.Create(propertyname,dict)
Prop.Create(propertyname,?attribute=value?,...)
```

Create a new property definition, and optionally set attributes of the new property.

The name of the new property must conform to the general property naming conventions in the toolkit. Indexed fields, instance identifiers, registration ID and other name parts aside from the base name are ignored.

In case a property definition for the same property name is available, and either already loaded or can be found by in the property autoloader path, the command internally succeeds, but still returns a scripting language error. This is intended to prevent the accidental overwriting of system property definitions, which can lead to problems which are difficult to detect and may lead to crashes, for example when the data type of an internally used property is changed in an incompatible fashion. Acceptance of the command when overwriting is desired can be forced by surrounding it in a **TCL** **`catch`** or **PYTHON** **`try/except`** statement. In the **PYTHON** case, the simple constructor always succeeds (due to lack of language features to support partial failure in the constructor), while the factory function **`Prop.Create()`** works like the **TCL** equivalent.

A property created without additional attributes has no associated functions, is of data type *string* and generally has empty default values for all attributes. The only exception is the **URN** namespace, which is set to the local namespace, and the creation date, which is set to the current time. In addition, the property object class is automatically set from the prefix of the specified name.

The processing of the attribute/value pairs is equivalent to a **`prop set`** command. The possible attributes are described under that subcommand.

Creating a property for which already a loadable definition exists in the installation environment results in this definition to be read first. The creation attributes then overwrite the values of the existing definition. In order to create a new property with all default fields, use the **`prop xcreate`** variant.

Example:

```
prop create A_MYPROPERTY datatype int author "A. Nonymouse"
```

This statement creates an atom property (determined by the standard `A_` prefix) of data type integer and a curious author attribution.

prop defined

```
prop defined propertyname ?alloworiginalname?
Prop.Defined(property=,?alloworiginalname=?)
```

Attempt to resolve the specified property name. If the property name is not yet part of the internal property database, an attempt is made to resolve it via the auto-loading mechanism. This is different from the similar **`prop exists`** command. The boolean return value is zero if the property name could not be resolved, one otherwise.

If the *alloworiginalname* flag is set, the property lookup will also use original names (such as SD file field tags) for identification. By default, only native toolkit names are recognized.

`prop available` is a deprecated alias of this command.

prop delete

```
prop delete propertylist ?force?
p.delete(?force=?)
```

```
Prop.Delete(properties=,?force?)
```

This command deletes zero or more property definitions from memory. It does not remove the files the property is defined by, if these exist. Property names which are not defined when the command is run are silently ignored. By default built-in properties cannot be deleted. In order to force this, the optional boolean parameter must be set. Deleting built-in properties is risky and can lead to crashes if the property is required as part of internal computations.

The property is not actually removed from the internal database until all instances of data of that property attached to chemical objects have been deleted. In such a case, the property name is not usable any longer in script commands, but implicitly information such as data type and formatting instructions are still used until all data associated with the original definition has been purged. The name of a half-deleted property still shows up in commands like **`ens props`**, but commands such as **`ens get`** fail, even for existing property data because the translation of the property name in the request to the internal data structure is no longer possible. However, operations such as file I/O which do not rely on properties looked up by string name may still succeed. The deletion of chemical objects with data from deleted properties always succeeds, decrements the reference counts and frees memory.

It is possible to re-define a property with the same name as a deleted one, even if there is still pending data. However, this is discouraged and potentially confusing.

If the supplied property name is an alias, the property the alias is pointing to is deleted, not the alias definition (see **`prop unalias`** for this function). In addition, all alias definitions which point to the deleted property are also recursively deleted.

The command returns the number of properties successfully deleted.

prop dup

```
prop dup propertyname newpropertyname
p.dup(newpropertyname)
Prop.Dup(propertynamename/pref,newpropertyname)
```

Create a new property definition using an existing definition as a template. In typical cases, the property duplicate is further customized by **`prop set`** commands. The new property name should not be the name of an existing property, though this is not illegal. In that case, it hides the definition of the original name similar to an alias definition.

The return value of this function is the name of the duplicated property.

prop exists

```
prop exists propertyname
Prop.Exists(propertyname)
```

This command checks whether a property is currently loaded in the in-memory database. No attempt is made to look up the definition via the auto-loading mechanism if it is not yet present. The boolean return value is zero if the definition is currently unknown, one otherwise. For a related command with implied auto-loading, see the **`prop defined`** command.

prop get

```
prop get propertyname attribute
p.get(attribute)
```

```
p.attribute
p[attribute]
Prop.Get(pname/pref,attribute)
```

Get the value of a property attribute. The property attributes are explained below in the paragraph dealing with the **prop set** subcommand. The return value is the current setting of the requested attribute.

The *parameters* attribute is a special case in multi-threaded scripts. Its value is thread-local. Every thread may have a different parameter set for each property. In order to obtain the base parameter set, the attribute *globalparameters* may be read.

prop getparameter

```
prop getparameter propertyname parametername ?usedefault?
p.getparameter(parameter=,?usedefault=?)
Prop.Getparameter(propertyname/pref,parameter=,?usedefault=?)
```

Get the value of a computation function parameter, or of a computation function default parameter. If the last command argument is not specified, or set to a boolean *false*, the queried parameter collection is the current parameter set (property attribute *parameters*). Otherwise, it is the default parameter collection (property attribute *defaultparams*). If the requested parameter is not found in the parameter set, an error results. Otherwise, the result of the command is the retrieved parameter value.

In a multi-threaded environment, a returned parameter value (but not a default parameter value) is thread-dependent. Every script thread has an independent property computation parameter set which is copied from the global value in the base interpreter the first time a property is referenced in a thread.

This command may be abbreviated to **prop getparam**.

prop getparameterdict

```
prop getparameterdict propertyname parametername
p.getparameterdict(parameter=)
Prop.Getparameterdict(propertyname/pref,parameter=)
```

Get the full definition of a property parameter, in addition to its current value. The command returns a dictionary which contains the following entries:

• *name*	The parameter name
• *description*	Free-form parameter description
• *label*	Parameter label
• *enumeration*	Symbolic values and aliases thereof, in standard enum format
• *defaultvalue*	Default parameter value as string (not included if undefined)
• *basevalue*	Current parameter value in base thread as string
• *value*	Current parameter value in current thread as string
• *minvalue*	Minimum value as string (not included if undefined)

• *maxvalue*	Maximum value as string (not included if undefined)
• *regexp*	Allowed value regular expression (not included if undefined)
• *xname*	Extended name
• *datatype*	Parameter value data type
• *constraints*	A list of active constraint flags, *none* if none are set
• *parsed_defaultvalue*	The default value parsed as the parameter data type
• *parsed_basevalue*	The base value parsed as the parameter data type
• *parsed_value*	The current thread value parsed as the parameter data type
• *parsed_minvalue*	The minimum value parsed as the parameter data type
• *parsed_maxvalue*	The maximum value parsed as the parameter data type

Parsed value entries are omitted if their corresponding string value is not defined, or the parse with the datatype-specific decoder fails. Parsed values are returned with the normal mapping to scripting language types (e.g. integer values if the parameter type is an integral numeric type), all other items are strings.

`prop getparamdict` is a command alias.

prop isdefault

```
prop isdefault propertyname value
p.isdefault(value)
Prop.Isdefault(propertyname/pref,value)
```

Check whether a property data value is the same as the property default value. The return value is the boolean test result.

prop ismagic

```
prop ismagic propertyname value
p.ismagic(value)
Prop.Ismagic(propertyname/pref,value)
```

Check whether a property data value is the same as the property magic value. The return value is the boolean test result.

prop isnull

```
prop isnull propertyname value
p.ismagic(value)
Prop.Isdmagic(propertyname/pref,value)
```

Check whether a property data value is the same as one of the potentially multiple defined **`NULL`** values.The return value is the boolean test result.

prop list

```
prop list ?pattern? ?constraints?
Prop.List(?pattern=?,?constraints=?)
```

List all currently loaded property definitions, including built-in definitions. If desired, a string pattern filter can be supplied to select only specific properties. The optional fourth parameter can be set to a combination of the bit flag values *computable* and *testable* (or *none*, if an explicit unset value is desired). These flags restrict the output to properties which have a *true* value in the attributes of the same name.

Example:

```
prop list E_* 1
```

lists all currently defined and loaded ensemble property definitions which are computable. Computability is only checked at the property definition level. It is no guarantee that the property computation succeeds for any specific ensemble or other chemistry object.

The return value is a string list for **Tcl**, or a reference list for **Python**.

prop query

```
prop query keyword ?objectclass? ?mode? ?casesensitivity?
Prop.Query(keyword=,?objclass=?,?mode=?,?casesensitivity=?)
```

Search the internal property database by matching the keyword against a standard set of property attributes, such as name, description, keywords, category, comment and **UUID**s. Only the current memory database is checked, no auto-loading or repository checks are performed.

By default all property definitions are matched. The object class argument (such as *atom*) can be used to limit the search to properties for a specific object class. Providing an empty argument is the same as omitting the argument.

The optional mode argument changes the string comparison mode. The default is *equal*, other possibilities are *substring*, *left* (match beginning of string), *right* (match end of string), *like* (as the **SQL** operator), *glob* or *regexp*.

The final argument can be *case* (case-sensitive matching) or *nocase* (case-insensitive comparison, this is the default).

The return value is a list of the version **UUID**s of the matched properties.

prop read

```
prop read filename ?doinstances?
prop read dirname ?doinstances?
prop read all
Prop.Read(filename=,?doinstances=?)
Prop.Read("all")
```

The first command variant reads a file with one or more property definitions. All property definitions found in the file are added to the internal in-memory database. By default, or when the *doinstances* parameter is set to a *false* boolean value, a definition from the file which refers to the same property as one already loaded implicitly deletes the old definition (see **`prop delete`** for a discussion of the consequences of this operation) from memory before the new definition is added. If the optional argument is set to a *true* boolean value, an additional definition with an instance number one higher than the highest currently defined instance is created instead.

The second command variant identifies all property definition files in **XML**-based or old keyword-based formats in the specified directory and loads them, without enforcing a specific load order. In the **PYTHON** variant, the argument name is always *filename*, regardless whether the argument value is a file or directory name.

The third variant traverses all directories in the property autoloader path and executes a directory property read, like the second command version, on all readable directories.

The return value of the command is a list where the first element is the total number of property definitions read from the file or directory, and the second element is the name of the first read property.

prop readblob

```
prop readblob data
Prop.Readblob(data=)
```

Decode a property definition stored in the data argument without the need to access the file system. The data argument is expected to contain a single **XML**-style property definition record.

If the decoding succeeds, the return value is the name or reference of the read property.

prop ref

```
Prop.Ref(propertyname)
```

PYTHON-only method to get a reference of the property, which allows terser attribute retrieval commands and other operations.

prop reload

```
prop reload propertyname
Prop.Reload(property=)
```

A convenience command for property computation module development. It first attempts to delete the current property definition. If this fails because its reference count is not zero, and attempt is made to reach zero by deleting all objects which may contain property data of the object class the property is associated with (e.g. ensembles if the property is an atom, ring or ensemble property). If the deletion now succeeds, the property is auto-loaded again, presumably with an updated processing script or module. Specifying an unknown property, a property where the reference count cannot be decremented to zero, or for which the definition cannot be found in the property loader path is an error.

The command returns the property *name*, even in **PYTHON** (i.e. no reference).

prop set

```
prop set propertyname ?attribute value?...
prop set propertyname dict
Prop.Set(propertyname/pref,?attribute,value?,...)
Prop.Set(propertyname/pref,dict)
p.set(?attribute,value?,...)
p.set(dict)
p.attribute = value
p[attribute] = value
```

This command is used to modify the attribute sets of existing property definitions. If the property name cannot be resolved, an error results. The following attributes are currently supported:

- *access*
  This argument controls the general access to property computation servers which accept remote requests. This is an enumerated property with possible values *none* (unset, no access by anyone if running as server), *private* (access only if client user is the same as server user, and client and server are on the same subnet), *local* (client user must be valid on local host, and client and server are on the same subnet), *registered* (client user must be in net group named *propname*_users), and finally *free* (no first line access checking on user IDs, subnets or net groups). If this access condition lets a request pass, script-based access checking still applies. This attribute has no effect on property computations which do not operate over a client/server connection.It has no effect in standard local scripts.

- *accessfunction*
  This name of an access check procedure if the property is computed on a server and remote requests are accepted. If there is no access function, only rudimentary access control (see *access* attribute) is performed. If an access function is set, it is called with the handle of the chemistry object the computation is requested for, and the name of the property. Additionally, request information including user name, email, host and password are stored in the global array variable `::server`. The access function may reject requests based both on the variable contents (wrong password, etc.) or the data of the chemical object the computation is requested for (too many heavy atoms, etc.). In order to reject a request, the access function can either return a boolean *false* value, or throw an error. Currently, the access check function is always executed in the global interpreter. Different from the computation, verification and configuration **Tcl** script functions, the access function is never automatically integrated into property definition files written by the **`prop write`** command. This function is intended to remain confidential and local. The access function is only called if a computation request is executed in a client/server relationship. It has no effect in standard local scripts.

- *address_city*
  The city part of the author contact address.

- *address_country*
  The country part of the author contact address, following the ISO3166 standard.

- *address_state*
  The state part of the author contact address. Empty if not applicable.

- *address_street*
  The street address part of the author contact address. Includes floor, house number, etc.

- *address_zip*
  The **ZIP** code or other applicable postal code of the author contact address.

- *affiliation*
  The institution the author of the property definition works for.

- *affiliationduns*
  The **DUNS** registration ID of the affiliated institution. This is primarily useful for US government projects.

- *affiliationurl*
  The **URL** of the affiliated institution.

- *altdatatypes*
  A list of alternative data types the property functions can handle. For example, most image types such as `E_GIF` or `E_EPS_IMAGE` can be computed both as datatype *diskfile* and *blob*. If a list of alternative data types is set, the property can only be reconfigured to use one of these (for details refer to the *datatype* attribute). If no list of alternative data types is specified, the data type can only be changed between types which use the same internal representation (e.g. *float* and *double*, or *int* and *short*).
- *altmodule*
  The name of an alternative structure representation interface module. Properties which have this attribute set are not computed on the normal internal **CACTVS** data structure, or by calling an external program by means of a script, but are linked with a third-party software component and interface code to translate between the data structures of **CACTVS** and the other program. This attribute is rarely encountered in normal use scenarios.
- *application*
  The name of an external application which generates the property data. This is a free-form string.
- *applicationdate*
  The release or version date of an external application which generates the property date. Must be a valid date specification.
- *applicationversion*
  The version of an external application which generates the property data. This is a free-form string.
- *attachment*
  This is a deprecated alias name for the *objclass* attribute.
- *author*
  The author of the property definition. This is a free-form string.
- *authorurl*
  A **URL** with information on the author, or an empty string if unset.
- *auxdata*
  This is the name of a file with arbitrary additional data the property computation function may need to perform its duties. When a property definition file is written with the **`prop write`** command, this data is encapsulated in the definition file. When the definition file is later read by an application, the data is written out to a temporary file, and the file name returned for this attribute is the name of the temp file, not the original file name. By this mechanism, it is not required to supply auxiliary data files as a separate files which may be difficult to locate and are easy to get lost. If this field is empty, no auxiliary data file is present.
- *category*
  A category string to be used if the property definition is stored in a repository.

- *charset*
  The character set this property uses for encoding and decoding. While this attribute is currently maintained as a free-form text field, it should only be set to values in standard character set syntax, such as *ISO-8859-1*. The latter value is the default in case it is not set.

- *checkfunction*
  The name of a scripted **Tcl** or **Python** procedure designed to verify that a value of a property is correct. This function is called with the property name and a chemistry major object handle as arguments. It should return 1 if the property data is set and the check succeeds, or 0 otherwise. It should not attempt to compute the data if it is not present. If the function can be found in the private slave interpreter of the property, it is executed there. Otherwise, an attempt is made to locate and run it in the main interpreter. When a property definition file is written with **`prop write`**, and the function can be located, it is encapsulated in the definition file and is available for all further interpreters which read the definition file. Example:

  ```
  prop check E_CAS $eh
  ```

  This command checks, if property `E_CAS` is set for the ensemble, whether it is formatted correctly by calling the **Tcl** check procedure defined for `E_CAS`. With this example property, the check function runs the **CAS** check digit algorithm on the data value. It catches cases where the re-computed check digit does not match the one found in the data.

- *classuuid*
  The base class **UUID** of this property.

- *comment*
  A free-form text comment.

- *computable*
  A read-only boolean status code indicating whether this property is in principle computable. A true status indicates that the property has a computation function and it is general suitable character.

- *computefunction*
  The name of the computation function. For computation functions implemented as compiled code, this data is for information only and cannot be changed in scripts. For scripted functions, this is the name of the **Tcl** of **Python** function which performs the computation. The function is called with the handle of the chemical object as parameter. In case the computation succeeds, the function should attach the property data to the object. In case of an error, an error condition should be raised in the script.

  If the function can be found in the private slave interpreter of the property, it is executed there. Otherwise, an attempt is made to locate and run it in the main interpreter. When a property definition file is written with **`prop write`**, and the function can be located, it is encapsulated in the definition file and is available for all further interpreters which read the definition file. Example:

  ```
  ens get $eh E_CAS
  ```

  This request calls the **Tcl** script function **`CSgetE_CAS`** which is defined in the property definition file as computation function for `E_CAS`.

  Properties which cannot be computed have an empty compute function name.

In case of **Tcl** script computation functions, all default and current computation parameters are automatically copied to the global (to the property slave interpreter, not visible in the main interpreter) array **`::params`**. This is done by first traversing the default parameter list and then the current list, so that parameters in the current list overwrite any default values. In addition, the current property computation time-out value is stored in **`::params(timeout)`**. Alternatively, the parameter values can be queried by **`prop getparam`** commands in the function. In multi-threaded scripts, each thread interpreter sees its private version of property computation parameters. Therefore, it is possible to compute different versions of the same property in parallel threads.

- *configfunction*
  The name of the configuration function for the property. This is a **Tcl** script function which is supposed to present a **Tk** panel with configuration options for that property. The *apply* function associated with the panel should modify the current parameter set of this property. The function is called with the property name as only parameter. Different from computation and check functions, this function is always executed in the main interpreter, in order to avoid the need to extend the slave interpreter into a full **Tk**-enabled interpreter with much larger overhead. When a property definition file is written with **`prop write`**, and the function can be located, it is encapsulated in the definition file and is available for all further interpreters which read the definition file. Example:

  ```
  prop configure E_GIF
  ```

  This command calls the configuration function for property `E_GIF`, which attempts to display a pretty large **GUI** panel with lots of sliders and dials allowing the user to adjust the rendering parameters.

  Properties which cannot be configured graphically have an empty configuration function name.

- *constraints*
  A set of flags to impose additional constraints on property values. Attempting to set a property value which does not meet these constraints in a script raises an error, or the value may be implicitly modified to fit the constraint. The check only applies to string conversions and conversions from **Tcl** or **Python** script interpreter objects. It is still possible to set a value in violation of the constraints by rogue compiled code in computation functions and by similar means. Not all flags are implemented, or make sense, for all data types. Implementing constraint checks is the responsibility of the data type handler module. Flags not supported for a specific data type are silently ignored. Any combination of flags can be supplied as a list argument. The currently supported values are

  - *none*
    no constraints, other than the native capabilities of the underlying data type
  - *notnegative*
    no negative numerical value
  - *notpositive*
    no positive numerical value), *notzero* (no zero value)
  - *nolowercase*
    no lower-case characters)

- *nouppercase*
  no uppercase characters
- *trimmed*
  leading and trailing white space are removed
- *nowhitespace*
  all white space is removed
- *odd*
  numerical value must be odd
- *even*
  numerical value must be even
- *notempty*
  input cannot be an empty string)
- *maptolowercase*
  translate to lower case
- *maptouppercase*
  translate to upper case
- *mapwhitspace*
  all white space sequences are translated to a single underscore character
- *no8bitchars*
  characters cannot have 8th bit set
- *maptotitle*
  translate to title formatting, with first letter of every word in upper case and all other letters in lower case

Example:

```
prop set E_ID constraints {notzero notnegative}
```

Assuming that `E_ID` has a numerical data type, any attempt to set it to 0 or a negative value in a script will fail.

Another method to apply constraints to property data is by means of the *regexp* and *regexpflags* attributes. Both methods may be used in combination.

- *cost*
  This is an integer parameter which is intended to give a rough estimate of the cost involved in computing a property. The default value is 1. Currently, its only use if when merging ensembles with different sets of property data. Here, no attempt is made to compute properties which have a cost of more than 5 and are only present on one of the ensembles.
- *ctxkey*
  The tag for this property in the **`ctx`** file format. If it is not set, a reasonable default tag is derived from the property name. This attribute is deprecated, the more general *filetags* attribute is the recommended replacement.

- *datadirectory*
  The installation-dependent directory path where property-specific data files are expected to be stored. The directory path is constructed from the installation path, the *propdata* subdirectory name and the property name in lowercase. Properties shipped with the toolkit adhere to this storage scheme for auxiliary data files. If the data directory does not exist, which is the case for properties which do not use external data files, the attribute value is an empty string. This attribute is read-only.

- *datamutex*
  Lock (with value 1) or unlock (with value 0) the mutex protecting the external property data directory (see *datadirectory* attribute). This is a write-only attribute. If a property intends to read or modify contents of its data directory, it should lock and then, after completion, unlock the mutex, to make sure modifications by other threads do not interfere. If the mutex is currently locked by another thread, the command will pause until the other thread has released it. Currently, use of the mutex to protect the data access is optional, but strongly suggested.

- *datatype*
  The attribute sets the data type of the property. The exact set of possible data types varies. Besides the rich set of built-in data types, an attempt is also made to locate a property handler module with the specified name in case the argument cannot be resolved with the built-in and currently loaded I/O handler modules. The default value is *string* - a simple 8-bit ISO 8859-1 string property.

  It is not possible to change the data type of a property if data instances of that property exist on chemical objects in the current application, and the internal representation of the old and new type is not identical. For example, it is possible to switch between *bool* and *int* types, but not between *int* and *string*, if active data of the old type has already been stored.

  The *altdatatypes* attribute imposes additional restrictions on possible values of this attribute.

- *date*
  This attribute stores the date of the property definition. It is used for information purposes only. If a new property is defined, it is initialized to the current date.

- *dbflags*
  This attribute encodes a set of hint flags for setting up **SQL** database columns for storing data of that property. The argument is a list of words representing bits which should be set. This attribute is intended to be used with application scripts and is not stored in property definition files. Various **SQL** statement construction functions will use information stored here for table and column definitions. It is a thread-local attribute. The following flags are currently supported:

  - *none*
    the default, no special treatment, *key* - property data will likely be used as query key,

  - *primarykey*
    column is a primary key for the table

  - *notnull*
    disallow the storage of `NULL` values for that data

    - *indexed*
      set up index on the property column,
    - *fixedlen*
      set up database column as fixed-width data, with actual width stored in the *length* attribute
    - *timeonly*
      for *datetime* data type, store only time part
    - *dateonly*
      for *datetime* data type, store only date part.

- *dbmap*
  Set the database, table and column mapping attribute of the property. On retrieval, trhis is a string in the format *database.table.column*, with unset parts omitted. On setting, the attribute may be set a a simple string (sets column name only), a string with a single dot (sets table and column) or a full string. This attribute combines the *dbmap_database*, *dbmap_table* and *dbmap_column* attributes. The property database map is thread-local. It is intended to be used to work with different databases in application scripts. It is not stored in the property description file. If a database map is set, various **SQL** statement construction functions will use this information.
- *dbmap_column*
  Set or retrieve the column part of the current property database map.
- *dbmap_database*
  Set or retrieve the database part of the current property database map.
- *dbmap_table*
  Set or retrieve the table part of the current property database map.
- *default*
  This attribute defines the default value for the property. Usually, the syntax for this item is the same as setting a specific data value, but data type handlers can set up a special default value decoder function. For example, array types allow a syntax like 10:0 for a default value of 10 elements set to 0 each, but this syntax is not supported when setting individual data instances.

  In case of properties of complex multi-field structure, the default value is a list of values in the order of the field definitions. The the default value list is shorter than the field list, the remaining fields have a zero/empty default value.

  Property data values which are identical to the default are not written explicitly in native **CACTVS** binary files. If such data is restored, and the default value has changed, the input data assumes the new default value because it is set from the current property definition, not the file contents.

- *defaultparams*
  The default set of parameters, as a keyword/value list or dictionary. The syntax is the same as that of the *parameters* attribute. This attribute set is intended to be used as a quick way to reset the normal parameter set after changes have been made. Example:

  ```
  prop set $p parameters [prop get $p defaultparams]
  ```

- *deinitfunction*
  The name of a de-initialization function to be called when the property definition is deleted. This is useful for example to remove data which is automatically set up on initialization and then held for the lifetime of the property definition. The type of the function is expected to be the same as that of the compute function, e.g. compiled code for **DLL/SharedLib** functions, script code for scripted computation functions. The function is called with two arguments, the property definition pointer and error function for compiled code, and the property name as only argument for scripts.

- *deletefunction*
  This is the name of a **Tcl** or **Python** function which is called whenever property data instances are deleted. The only argument to the function is the value of the data item. This function may perform custom clean-up operations, such as taking down editing forms.

  If the function can be found in the private slave interpreter of the property, it is executed there. Otherwise, an attempt is made to locate and run it in the main interpreter. When a property definition file is written with `prop write`, and the function can be located, it is encapsulated in the definition file and is available for all further interpreters which read the definition file.

- *depends*
  A list of properties which this property depends on as input data when it is computed. This information is used to automatically invalidate or re-compute data instances when underlying, more basic data changes on the chemical object. This is a recursive process. The invalidation of one set of properties is cascaded to all property instances on the chemical object which themselves depend on the just invalidated data. An update cascade can be triggered for example implicitly by statements such as `atom set` or `ens new`, or explicitly by `ens taint`. Note that simple data deletion commands such as `ens purge` are not triggers. Property data can be prevented from auto-deletion by using lock statements such as `ens lock`. The updated data which acts as the trigger is never itself deleted, even if a dependency cycle exists.

  For this argument, it is explicitly allowed to set dependency property names which are not yet defined. No attempt is made to resolve these, but every loading or creation of a new property definition in another context will lead to a regeneration of the pre-parsed form of dependencies on all current internally held property definitions, potentially adding more, or different, dependency links. Property names which cannot be resolved are silently ignored.

- *description*
  This is a free-form string which describes the property.

- *displaywidth*
  A default value for the width of a display column, measured in characters. The default value is negative, meaning that scaling is dynamic.

- *doi*
  A digital object identifier for the property, if defined.
- *email*
  The email address of the creator of the property definition, useful in case a definition file is distributed and anybody has questions.
- *enum*
  This attribute defines enumerated symbolic names for specific property values. They can be used for mnemonic data input, and many output modes also use these names by default (cf. **`ens get`** vs. **`ens nget`**). Input of enumerated values is always case-insensitive. It is not illegal to store data values of an enumerated property which are not represented by a corresponding symbolic name. These values are simply encoded as or decoded from numerical values.

  For properties which consist of multiple independent sub-items, or sub-item data types, the enumeration is a list. The list is split, and each element applies to a different field in the order of field definitions. Examples for this are the *compound* and *choice* types, but not simple vectors. For simple vectors, the enumerated names apply to each element.

  The enumeration is a string, where blocks corresponding to a specific value are separated by a colon. In the absence of an explicit value designator, the first block corresponds to value zero. Multiple words separated by commas are interpreted as alias names. On output, the first word is used as the canonic designator. Example:

  ```
  prop set E_CLEARANCE enum \
  "public:private:topsecret,destroy_before_reading"
  ```

  The enumeration defines symbolic names for value 0 (*public*), 1 (*private*) and 2 (*topsecret*, or alias *destroy_before_reading*). On output, a data value 2 is always displayed as *topsecret*.

  There are a couple of additional syntax elements. A code which is not in the automatic value sequence can be specified with a *=n* construct. The next value block, if it has not an explicit value designation of its own, is the value of the previous block plus one. Here is the enumeration for property `A_LABEL_STEREO`, with codes for standard parities and square planar stereochemistry:

  `M,-=-1:undef=0:P,+=1:U,C=2:Z,N=3:X=4`

  The normal 0,1,2... implicit value sequence can be changed to a power-of-two sequence by prefixing the enumeration string with a “^” character. In that case, the implicit numerical value o the blocks are zero, one, two, four, etc. For bit types (64-bit set and bit vector), this interpretation of the enumeration string is automatic and does not need to be spelled explicitly. Output of data items for properties with a bit-based implicit or explicit enumeration string consists of a set of all words with corresponding set bits, and one possible input format is a list of words for all bits that are set.

  A final syntactic shortcut is the use of the “@” character as prefix. In that case, the enumeration is interpreted as a symmetric value set centered around zero. If there are three value blocks in the enumeration string, and no explicit value designations, the encoded numerical values are, from left to right, minus one, zero and plus one.

  Characters with special meaning in parsing enumeration strings (equal, comma, colon) cannot be part of a symbolic name. Prefix characters may be used in enumeration names, but not as part of the first word.

- *fielddatatypes*
  A list of the field data types defined for the property. This is a subset of the information encoded by the *fields* attribute, and read-only.

- *fieldindices*
  A list of the choice indices of the field definitions. This attribute is useful only for *choice* and *choicevector* properties. This is a read-only attribute.

- *fieldnames*
  A list of the field names defined for the property. This is a subset of the information encoded by the fields attribute, and read-only.

- *fieldproperties*
  A list of the secondary properties associated with the fields of the current property. Fields which are not property-associated report an empty element. This is a subset of the information encoded by the fields attribute, and read-only.

- *fields*
  This attribute describes fields of a property. Its exact meaning depends on the data type of the property. The attribute is a nested list. Each list element is a list which contains a field name, its data type (optional, defaults to *string*) and its unit (optional, defaults to *undefined*). Simple properties which do not have sub-items do not possess a field set. For simple vectors, field names may be defined to facilitate convenient vector element access, but they have no effect on the internal representation of the property values. Example:

  ```
  prop set A_XYZ fields {x y z}
  echo [atom get $eh $label A_XYZ(x)]
  ```

  The code above shows you how to name elements in a simple vector, and how to use them conveniently for retrieval. For the *bit* and *bitvector* data types, such fields are interpreted as individual bits.

  However, the field set is essential for the definition of properties with a complex internal structure, such as data types *compound* and *choice*, and the vector variants thereof. For these types, the field set should be set up when the property is defined, and not changed at any time later. Example:

  ```
  prop create E_MYCOMPOUND datatype compound \
  fields {{name string} {fval double angstrom} \ {timestamp date}}
  ```

  The statement above defines a compound property with three fields of different data types, one of them with a standard unit. Another use is for the definition of *choice* properties, which hold one of several possible variants of data, possibly of different data types:

  ```
  prop create E_MYCHOICE datatype choice \
  fields {{name string} {ival int} {timestamp date}}
  ```

  The difference is that the example compound property contains three parallel fields of different data types, while the choice property has just a single field, but it can be of one of three different data types at any time. For the compound example, the string input or output form of a property value is a list of three values, where each element is parsed or formatted according to the data type of the field. For the choice example, the string format is a single two-element list, where the first element is the field name, which implicitly defines the data type, and the second element the value. Another example:

  ```
  prop create E_MYCOMPOUNDVEC datatype compoundvec \
  field {{name string} {ival int} {timestamp date}}
  ```

This is the specification for a compound vector. Each vector element is a compound of three data items. It could be set to a vector with two elements with a statement like

```
ens set $eh E_MYCOMPOUNDVEC \
{{“name1” 1 now} {“name2” 2 tomorrow}}
```

In the same fashion, choice vectors may be defined and set. A choice vector has a single data item per element, but each of these elements can be of a different data type from among the selection defined in the field set.

In addition to having a simple basic data type, the fields of complex data types can also be indirectly defined by a reference to a secondary property, which defines the structure of that field. These indirect references can be of complex types themselves. If the secondary property has a defined unit, this unit automatically sets the field unit, too. Example:

```
prop create T_NCBI_PUBLICATION_PATENT_ID \
datatype compound \
fields {{country string} {id T_NCBI_BIBLIO_PATENT_ID} {doc-type string}}
```

This compound property has three fields. The first and second are simple basic types, but the third is a complex property of type *compound* with its own fields, which potentially could themselves be recursively defined by more property references. The string representation of data for this definition is a list of three elements, where the middle element is itself a (potentially deeply nested) list.

If an input list for a complex property is shorter than the size of the field list, the rest of the items are set to their respective default values.

If the data type of a field cannot be decoded from the information currently in memory, an attempt is made to auto-load the data type I/O handler or, in case of property references, a property definition. If that fails too, an error occurs. Data types must be specified in lower case. This is needed to distinguish them from property names, which are upper case.

In case of symbolic names assigned to vector fields, the data type of the field is ignored and can be omitted, since it directly follows from the vector definition.

The native **CACTVS** file formats are designed to cope with complex property data for which the field definitions have changed between the time the file was written and when the data is read again. Every field is stored with its explicit name and data type. If the current property definition has additional fields compared to what is stored in the file, these fields are initialized to the default value. Fields which are no longer present in the property definition, but whose data type can be decoded from the information in the file are skipped and ignored.

If a **`prop get`** command is executed for this attribute, the output is a nested list. Each field description always contains the field name and the data type. If the property is of the *choice* or *choicevector* data type, the choice index of this field is appended next. Finally, there can be a property reference or unit element which is appended only if a secondary property reference or a standard unit is defined for the field. Ultimately, each field record can consist of two to four elements.

- *fieldunits*
  A list of the units of the fields defined for the property. Fields without a defined standard unit report an empty element. This is a subset of the information encoded by the fields attribute, and read-only.

- *fileflags*
  This attribute is only used when also a *fileformat* value is set (see below). In that case, this attribute defines the decoder flags used when reading the property data as a string file record in an **`ens create`** command.

- *fileformat*
  The name of a structure file I/O module associated with this property. The data of properties which possess this attribute are images of file records. Examples are `E_SMILES` (file format *smiles*, `E_SMARTS` (file format *smarts*) or `E_SDF_STRING` (file format *sdf*). This information is used when a structure is decoded from such data by means of the **`ens create`** command, and instead of a specific decoder mode the property name is supplied.

- *filetags*
  A dictionary of the tags to be used for storing property data in various file formats. Currently, only the *cml*, *cex* and *ctx* I/O modules make use of this information. The keys of the dictionary entries are the file format names, the values the associated tag strings. This attribute does not trigger the autoloading of the specified I/O modules at the moment the attribute is set. Rather, these tags are resolved or disabled when a matching I/O module is loaded or unloaded by another mechanism, for example explicit loading (**`filex load)`** or implicitly via a file suffix match. There can only be one tag per file format - the last is used in case of duplicates -, and the total number of different file formats a tag can be set for is ten per property, though the tag/file format sets of different properties do not need to be identical.

- *filters*
  This attribute is a list of filters which limit the definition range of a property to a subset of objects. Example:

  ```
  prop set A_FREE_ELECTRONS filters classicatom
  ```

  This statement, which just mirrors the actual definition of property `A_FREE_ELECTRONS`, tells the toolkit that the concept of free electrons is meaningful only for classic atoms with are an element, but not for super-atoms, query atoms and similar constructs which are subclasses of the atom minor object. In case there are multiple filters, the object must pass all of them in order to make a property data item defined. In case a specified filter name is not yet defined, an attempt to autol.oad the filter definition is made.

  Currently, there is a maximum of 10 filters per property definition.

- *flags*
  The *flags* attribute holds a collection of general flags which encode the status and operation mode of the property. This is an enumerated set. Currently supported flag names include

  - *autobackup*
    when property is changed via script, automatically save last values in backup property, i.e. `A_LABEL` in `A_LABEL%`

  - *expandpriority*
    if a chemistry object such as an atom which hold a data item of this property is replaced by, for example, an expanded fragment, this data is transferred to the first replacement atom of the expanded fragment, and not computed from the properties of the new atom.

- *fixedlength*
  the size of property data is constant, even for data types where each data item could have an individual size, for example vector types, or strings. This information is useful to optimize storage layouts in various contexts. The actual length of the data item is either taken from the *length* property attribute, or from a sample data item on a chemical object which is expected to be a representative with the correct length. The short form *fixedlen* is an alias-
- *globalexecution*
  execute computation script and other scripts associated with this property always in the global interpreter, not a property-specific slave.
- *internaltimeout*
  indicates that the property computation function performs its own internal timeout handling. No timeout interrupt framework is installed when a computation of this property is invoked.
- *labelkey*
  this property is the minor object label key property for the object class it is associated with. For atoms, this is `A_LABEL`, for bonds, `B_LABEL`, etc. There can only be one such property per object class, and there must be one such property for all object classes which are not major objects. Changing the label key properties should only be attempted under extraordinary circumstances.
- *localupdate*
  the property computation function supports the update of a single specific object, not all objects of the same class linked to a major object (i.e. it can recompute data on a single atom and not just all atoms in the ensemble). This flag is for information only. Do not set it directly.
- *locked*
  if set, the property cannot be overwritten by reading property definition records describing properties with the same name. Individual attributes of the property may still be changed, including resetting this flag by script commands. This attribute is for example useful when the current definition of a property should be maintained after opening a *cbs* or *bdb* file - these include the property definitions used at the time the file was written.
- *mergedata*
  merge multi-line input data into a single data item, for example a string with tab separators. Has an effect only for input from multi-line **MDL SD** data fields.
- *mergedefault*
  if data of a non-computable property is present only on one object in an object merge operation, by default this property data is discarded in the process because there is no way to obtain it for the second object. If this flag is set, the object without the data is instead initialized with default data for the property, and the merge does not discard the original information from the object where it was present.
- *none*
  no flags are set. On setting, you can also pass an empty string.

  - *portable*
    the property has no OS-dependent components such as DLLs or shared libraries for the computation module, or at least such components are available for all currently supported toolkit platforms.
  - *trace*
    for scripted properties, trace execution when the computation function is called.
  - *trusted*
    the property is trusted, relax usage restrictions and safety belt features.
  - *nominmax*
    property has no definition of, or use for, minimum and maximum property values. This can reduce memory usage and increase performance.
  - *nosighandler*
    run without signal handler to trap core dumps etc. even if property is not trusted. Signal handlers are expensive to set up and tear down, so if a function of an untrusted is called millions of times, this can significantly improve performance.
  - *notimeout*
    disable computation timeout watchdog, even if a timeout value is set.
  - *synthetic*
    the property was synthesized from minimal information without a proper property definition. Its name contains an asterisk to make this immediately obvious.This flag is for information only, do not set it directly.
  - *timing*
    measure execution time for each computation function call and store in property metadata record on the major object the computation was performed for.
  - *threadtrace*
    trace multi threading-related synchronization operations.
  - *uselookuptimeout*
    if the flag is set, and no property-specific global or thread-local timeout has been set, use the global Internet lookup timeout value in `::cactvs(lookup_timeout)` instead. This flag is typically used by properties where the computation involves retrieving data from Internet sources.

  The standard bit set manipulation prefixes (+,-,^) are supported. Example:

  ```
  prop set A_XY flags +notimeout|trace
  ```

  This command adds two flags to the current flag set.

- *format*
  Internal use only
- *functionsource*
  The source code of all of the script functions which are contained in the property definition. Code for functions which are not defined in the definition file, but for example reside in global library source files, are not returned when the attribute is queried. If the attribute is set, its contents replace the current function definitions.

- *functiontype*
  The type of computation function this property provides. Possible values are *none* (no computation function), *bultin* (built-in function), *dll* (dynamically loaded compiled module), *tclscript* (interpreted **Tcl** script), *syncserver* (synchronous client/server computation, **Cactvs** native **RPC** protocol), *asyncserver* (asynchronous client/server computation, **Cactvs** native **RPC** protocol), *mail* (send request and receive answer by email), *altrepmodule* (via an alternative data representation module), *default* (pseudo-function, a computation request always results in default values as result), *disabled* (temporarily disabled), *indirect* (computation happens as windfall by computing a different property) and *soap* (communication via **SOAP**-based service). **Python**-enabled toolkit versions additionally support the *pythonscript* function type, and versions with a **Java JVM** the *java* type.

  In most cases, the function type is defined once when the property is set up and not changed later. One potentially useful application of changing the function type is when a computation is farmed out to a public or access-controlled server, for example because a local optional computation module is missing. Example:

  ```
  prop set A_XYZ functiontype syncserver \
  host www.xemistry.com port 17890 password xxx
  ```

  After this statement, requests for atomic 3D coordinates will no longer be made by attempting to load a local module and execute the coordinate generation code on the client machine, but by communicating with the server specified above. this is a sample host name and not a real publicly accessible service.

  Setting some other attributes (such as the *indirect* property) implicitly updates this attribute.

- *globalparameters*
  This is the global version of the current parameter set. Its meaning is explained in the paragraph detailing the *parameters* attribute.

- *helpful*
  This is a list of source data properties which can be helpful for computing this property, but do not need to be present for the computation to succeed. This is for example used as a hint in network communication. If the property data is already present, it is sent, but no attempt is made to compute it, and the computation function will either do without these additional data, or compute it itself. This attribute has no effect for local computations. Its counterpart is the *required* attribute.

- *host*
  The name of the host to contact to request a remote computation for this property. This applies only to properties which are farmed out to servers and is ignored otherwise.

- *id*
  Get the internal integer ID code of the property. An ID value is automatically assigned when a property is created or, for built-in properties, on start-up. This is a read-only attribute and cannot be set.

- *indirect*
  If this attribute is set to the name of another property, the computation function of that other property is expected to compute the data for this property as a side effect. This property should be listed in the *windfall* attribute of the second property. When the computation of this property is requested in a script, the request is re-schedule as a computation request for the *indirect* property.

  This mechanism is useful because you only need to maintain a single computation module or script for a set of properties which are computed synchronously. One of the properties provides the computation function, and lists the other properties in its *windfall* attribute. The other properties from the group all simply refer via an *indirect* attribute to the single property with the computation function.

- *infourl*
  A **URL** with information on the property definition, or an empty string if unset.

- *invalidation*
  This set of flags describes under which circumstances the property data attached to an object is automatically discarded. The flag sets augments the property dependency relationships defined via the *depends* attribute. These are s global operations on a structure beyond the scope of isolated property data changes. With the exception of the *never* value, multiple code words can be usefully combined into a list. The currently supported flags are:

  - *never*
    no automatic invalidation
  - *atomchange*
    any major atom change in an ensemble, such as an atom type or element change, or the addition or deletion of atoms
  - *bondchange*
    any major bond change in an ensemble, such as a bond type or order change, or the addition or deletion of bonds
  - *stereochange*
    any stereochemistry change in an ensemble
  - *groupchange*
    any major group data change in an ensemble, similar to the *atomchange* and *bondchange* criteria
  - *merge*
    data of two objects is merged
  - *3dop*
    the 3D structure of an ensemble changes in a way that affects inter-atomic distances
  - *3dglop*
    the 3D coordinates of an ensemble change globally, but inter-atomic distances remain constant
  - *shuffle*
    the order of atoms changes in an ensemble

- *hadd*
  hydrogens are automatically added or removed to an ensemble
- *dup*
  a major object is duplicated.
- *atom*
  the atom list composition changes for an ensemble
- *bond*
  the bond list composition changes for an ensemble
- *mol*
  the molecule list composition changes for an ensemble
- *ring*
  the ring list composition changes for an ensemble
- *sigma*
  the sigma system list composition changes for an ensemble
- *pi*
  the pi system list composition changes for an ensemble
- *group*
  the group list composition changes for an ensemble
- *surface*
  the surface patch list composition changes for an ensemble
- *ringsystem*
  the ringsystem list composition changes for an ensemble
- *reaction*
  reaction membership changes for an ensemble, or the ensemble set changes for a reaction
- *dataset*
  dataset membership changes, either for an object in a dataset, or the dataset object proper for which the object set changes
- *file*
  the association of an object with a structure file changes, for either of the sides
- *table*
  the association of an object with a table object changes, for either of the sides
- *network*
  the association with a network object changes
- *vertex*
  the vertex list composition changes on a network object
- *connection*
  the connection list composition changes on a network object

Example:

```
prop set E_MY_UNIQUE_ID invalidation dup
```

Property data for `E_MY_UNIQUE_ID` is discarded from the cloned structure when duplicating ensembles.

A signal indicating that certain operations have taken place can be explicitly sent by the *taint* major object subcommands. Example:

```
ens taint $eh {atomchange bondchange}
```

Above command leads to the shedding of all property data on the ensemble which is not robust with respect to atom and bond edits. These events are always global on the controlling major object. It is not possible to signal an update event for a single atom or bond.

- *keycolumn*
  The name of a database column for keyed properties which are retrieved from a database-style source. In case of data sources which do not have a concept of columns, this is an empty string.
- *keydatatype*
  The data type of the access key of a keyed property. If the key is another property (*keyproperty* attribute), the data type is derived from the property definition of the key property and potentially its field index and should not be changed. For keys which are not properties, this attribute must be set. It can be any supported property data type, including those which need to be loaded as external handler module on demand.
- *keyenum*
  An enumeration similar to the *enum* attribute for the encoding and decoding of key values for keyed properties. It the key is a property, this attribute overrides the *enum* attribute the key property might possess itself in the context of key operations.
- *keyproperty*
  The name of a property which is used as access key to obtain property values by retrieval from an external data source. This must be resolvable to a property definition. If no key property is used, this is an empty string.
- *keyreference*
  The data source for key-based property retrieval, encoded as a **URN**. The protocol field of the **URN** determines the name of the key resolver used. An attempt is made to auto-load the proper key resolver module if the handler has not yet been loaded.The rest of the **URN** fields are interpreted by the resolver module. They usually follow the standard **URN** syntax with user IDs, passwords, hosts, ports, and a file or database table component, with reasonable defaults if any parts are not specified.
- *keywords*
  A list of keywords associated with the property.
- *length*
  The data length of the property. A value of -1 (the default) indicates that the length should be expected to vary. The exact interpretation of the attribute is dependent on the data type. For example, for strings and Unicode strings it is the character count, for blobs the byte

count, and for vectors of constant element size (for example, floating-point and integer vectors) the element count. If a length is set, it is only used as a hint for internal optimizations. It becomes a binding statement only if the *fixedlength* bit in the *flags* property attribute and/or the *dbflags* attribute has been set.

- *license*
  The license class associated with this property. Setting the license to a standard type updates the associated **URL** with a standard location.
- *licenseurl*
  A **URL** with details about the property license.
- *literature*
  This is a free-form string for a literature reference describing the meaning and/or algorithm behind the property.
- *logfile*
  In case the property logging mode is set to write to a file, this is the name of the file the log output is written to.
- *logmode*
  This attribute is an enumerated set of words which describe the handling of log or debug output during calls of the property computation routine. If any mode except *ignore* is set, standard output and standard error are redirected to the selected output channel before the function is called, and restored to the previous channels afterwards. The default should be *ignore*, because redirection involves several system calls and therefore can significantly slow down the computation of trivial properties. The supported values are:
  - *ignore*
    neither redirection nor restoration
  - *suppressed*
    redirection to `/dev/null or NUL on Windows`
  - *console*
    redirection to the `/dev/console` (not on Windows)
  - *filewrite*
    redirect to the file specified in attribute *logfile*. The file is deleted before each computation function call.
  - *fileappend*
    redirect to the file specified in attribute *logfile*. The output is appended.
  - *stdout*
    redirect both standard output and standard error to original standard output
  - *stderr*
    redirect both standard output and standard error to original standard error
- *magic*
  Define a magic value for the property. This is a convenient way to mark one specific property data value as an indicator for a special condition outside the normal validity range of the property. The attribute must be a parseable representation of a property value.

Magic values are not written out explicitly in native **CACTVS** binary files. If therefore the magic value changes, newly read magic property data values automatically assume the current magic value because they are set from the current property definition, not the file contents.

- *max*
  The maximum expected property value. This attribute is not enforced - it is primarily intended to be used to compute color scales etc. from property values. See also the *min* attribute. The attribute must be set to a valid string representation of a property value which can be decoded with the current property definition, and should not be the same as either the *magic* or *null* attributes.
- *menugroup*
  The name of a menu group the property should be sorted into under the name specified in *menuname*. This is only used in graphical user interfaces.
- *menuname*
  This is an alternative, user-friendly name to be used in menu displays and similar circumstances. This attribute is a free-form string. Since it is not decoded as property name, it does not need to adhere to the **CACTVS** property naming conventions.
- *mergefunction*
  The name of a **TCL** or **PYTHON** function which is called when property data attached to a major object is prepared for merging, for example in the context of the **`ens merge`** command. The type of the function must be the same as that of the compute function. In case this is a **TCL** or **PYTHON** procedure, it is called with the the two object handles (**TCL**) or references (**PYTHON**) as argument.

  If the merge function can be found in the private slave interpreter of the property, it is executed there. Otherwise, an attempt is made to locate and run it in the main interpreter. When a property definition file is written with **`prop write`**, and the function can be located, it is encapsulated in the definition file and is available for all further interpreters which read the definition file.

  The purpose of the merge function is to modify the data values in the second object in order to prepare its merge with the first object. It must not perform any merge operation itself. For example, the built-in merge functions for minor object labels (properties `A_LABEL`, `B_LABEL`, etc.) adds an offset to the label values of the second object in order to avoid the assignment of identical labels to two minor objects in the merged ensemble. The merge function for atomic 2D coordinates (property `A_XY`) scales the coordinates of the second object to fit with the scaling of the first, and then adjust the coordinates so that the structures are displayed side by side after merging.
- *mimetype*
  A helpful **MIME** type associated with the property. It is for example use to identify the format of contents of property data held in blobs or disk files. Commonly used values are for example *image/png*, *image/gif* and *image/svg+xml* to identify the encoding of image data. The default value is empty.

- *min*
  The minimum expected property value. This attribute is not enforced - it is primarily intended to be used to compute color scales etc. from property values. See also the *max* attribute. The attribute must be set to a valid string representation of a property value which can be decoded with the current property definition, and should not be the same as either the *magic* or *null* attributes.
- *name*
  This is the name of the property. Changing it after it has been defined originally is generally considered bad style, but it is possible. Internal data references use pointers, not strings with names.
- *nativename*
  Internal use only.
- *null*
  The string representation of the first **NULL** property value. Usually this is set to the same string as the *magic* attribute, or an empty string. The value must be a valid string representation that can be decoded with the current property definition if it is not an empty string. An empty string disables the *null* value check and is not parsed, even if that is a valid representation of a property value.
- *null1*
  This is an alias for the null attribute above.There are a maximum of three recognized **NULL** values - this follows the **SPSS** statistical analysis package data model. The second value is only applied if the first value is set, and the third only if the first and second are defined.
- *null2*
  The second null value.
- *null3*
  The third null value.
- *nullrange*
  A list with a pair of elements defining an inclusive lower and upper value limit for property values regarded as **NULL** data. The format of each of those elements is the same as for the *null* attribute described above. This range criterion is distinct from the normal *null* parameter - both may be used in combination. For properties where the associated property data type does not support larger/smaller comparisons, this attribute is not useful.
- *objclass*
  The name of a chemical object class the property is attached to. By default, this is automatically set by decoding the prefix of the property name, and it is highly recommended that you do not override this automatic prefix scheme. *attachment* is an alias for this attribute.
- *objectfile*
  This is the name of the shared object (DLL, Bundle) file for a dynamically loaded computation modules. This attribute is empty for properties which do not possess a compiled computation module.

- *originalname*
  The original name of the property. When data is read from a file, and the tag the data is stored under does not match the **CACTVS** property name, this attribute is automatically globally updated and set to the name as found in the file. There is currently no mechanism to remember multiple original names from different files in parallel.

  When property data is written to a file, and an original name is set for an output property, this name is used in preference to the **CACTVS** name if the file syntax allows it. For example, this is the attribute which should be changed in order to set arbitrary SD data field names for property data.

  This attribute can be set to arbitrary strings and does not need to follow the **CACTVS** property naming conventions. Nevertheless, names set via this attribute **are** used to resolve properties only as fall-back. The priorities for name resolution are aliases first, then proper **CACTVS** names, and finally original names. Menu names are never used for property lookup.
- *orcid*
  The **ORCID** code of the author of the property definition (see www.orcid.org).
- *outputlevel*
  Deprecated. Originally intended to allow the selection of groups of properties up to a certain level of importance in output sets without the need to name them explicitly. The higher the specified level for a property, the more important it is supposed to be. Its output level value is compared to the output level of, for example, a structure file handle, and the property is included if its level is equal to or larger than the file object level. In practical experience, this does not work too well. In any case, the decision to include or suppress the data of a property via this criterion can be overruled by the property *outputmode* attribute, and the write and suppress lists of structure file objects.
- *outputmode*
  This attribute is one of several attributes which control whether data for a property is included in output or not. It can be one of three values:
  - *suppressed*
    the data is never output
  - *standard*
    the property output is decided by other factors, such as the *outputlevel* attribute or property control lists of structure file objects
  - *forced*
    the property is always written, regardless of other settings.

  The default value is *standard*.
- *path*
  The repository path for displaying hierarchical repository trees. This attribute is independent of any file system paths.
- *parameters*
  This is a keyword/value list of the adjustable parameters used in the property computation routine. It must be set as a proper **TCL** list owith an even number of elements. or, equivalently, a **TCL** or **PYTHON** dictionary.

For multi-threaded scripts, this attribute is thread-local. When a script thread first refers to a property, its thread-local parameter set is copied from the current global setting. All further manipulations only change the thread-local copy. The only exception is the base thread, which updates the master copy. In case a script thread needs to change or query the global setting, it can use the *globalparameters* attribute. For the base thread, these two attribute names are identical. The split of the parameter set into thread-local versions allows multiple threads to compute different variants of data for the same property simultaneously.

Convenient methods to access the parameter set are via statements like

```
array set params [prop get $pname parameters]
set paramdict [dict create {*}[prop get $pname parameters]
```

The first example initializes the array variable **`params`** to proper element names and values. The second method initializes a **Tcl** dictionary.

The reverse path is also straightforward:

```
prop set $pname parameters [array get params]
```

Please refer also to the paragraph about the *computefunction* attribute for more details on how to access the parameter set in computation functions.

When used with **`prop set`** or **`prop create`** in the base thread, the more detailed typed parameter dictionary is replaced. All parameters defined this way are untyped and unrestricted strings.

- *paramdict*
  When used with **`prop get`**, report exhaustive parameter info in the form of a list of dictionaries. The outer list contains an element for every defined parameter, in order of the definition. Each list element is the full parameter dictionary of that parameter (see **`prop getparameterdict`** command). *parameterdict* is an alias for this attribute.

  When used with **`prop set`** or **`prop create`**, the full parameter set of the property is replaced, and the untyped simple parameter list rebuilt from the new detailed specification. Different from the *parameters* attribute, the effects of this command are not thread-local. The edits always apply to the base property definition.

- *parametercount*
  This is a read-only attribute which returns the number of currently defined computation parameters for this property. It is half the length of the list returned by reading the *parameters* attribute.

- *password*
  A free-form text password which is transmitted as part of the request data for computations which are performed on a remote server. The server may use this information to grant or deny the processing request. This attribute is not used for local computations.

- *phone*
  A contact phone number of the property author.

- *port*
  The port used to contact a remove computation server. The default port for distributed **Cactvs** computation servers is 16520. For local computations, this attribute is ignored.

- *precision*
  The number of significant digits after the decimal point for floating-point data.

- *precompute*
  This is a list of properties for which an attempt is made to pre-compute them before the computation function for this property is called. In case the computation fails, it is silently ignored.

  This attribute is primarily useful to control the location of computation for low-level property data when using server-based set-ups. In case the computation is performed implicitly in the computation function running on a server, the low-level computations also take place there if they are not themselves farmed out. If these properties are listed as *precompute* properties, they are computed on the client side before the request is sent.

- *profile*
  Internal use only.

- *regexp*
  This attribute provides a method to check the encoding of property data values that are read from strings, or from **Tcl** or **Python** interpreter objects. The argument is a regular expression. If input data does not match it, it is rejected. The check is only performed at the time of input via the handler function. Compiled code manipulating internal data structures directly can still produce data which does not pass the expression filter.

  Data format handler modules need to implement this function individually. At this time, the data types which support this are *string*, *stringevector*, *unicode* and *unicodevector*. Data format handlers which do not support regular expression filtering silently ignore it. The match operation of the regular expression can be further controlled with the *regexpflags* attribute.

  This attribute can be used in combination with the *constraints* attribute.

  Example:

  ```
  prop set E_REGID regexp {TinyPharma[0-9]+}
  ```

- *references*
  Cross references of the property definition. This is a nested list of class **UUID**s and reference type tags.

- *regexpflags*
  This set of flags controls the match condition for regular expressions set via the *regexp* attribute. It is a set of flags. The currently supported flags

  - *none*
    no flags, can also be set as empty string
  - *extended*
    use extended regular expression syntax
  - *nocase*
    ignore case

  Example:

  ```
  prop set E_REGID regexpflags {nocase extended}
  ```

- *regid*
  The official registration ID of the property. Properties of general usefulness can be registered and be assigned an official numerical identifier. This attribute must be set by a (potentially, digitally signed) property definition record. It cannot be set by script commands.
- *required*
  This attribute is a list of properties which need to be present before a remote or threaded computation of property data can be attempted. In client/server computations, the client tries to compute all these properties and send the data (together with helpful, but not mandatory data - see *helpful* attribute) to the server. In case of a threaded property computation, the required properties are computed on the original object before it is duplicated for use with the independent thread. This attribute has no effect on local computations.
- *scale*
  A scaling value for numerical data. The default is 1.0. This attribute is defined, but currently only passed on for **MDL XDF** file format output. It has no effect on the interpretation of property data
- *separator*
  A field separator which enables word-indexed addressing in string-style property data types. Example:

  ```
  prop create E_MYID separator :
  set eh [ens create $smiles]
  ens set $eh E_MYID "p345:c64587:u45"
  ens show $eh E_MYID(1)
  ```

  This returns “c64587”. In case the separator is more than a single character, any of the listed characters is considered a word boundary. The split occurs on any sequence of characters from the separator set. The default separator is white space.
- *sourcefile*
  The name of the source file of a compiled property computation module. No guarantee is given that the file is contained in any specific toolkit distribution.
- *status*
  A read-only attribute which lists the currently set flags in the internal status word of the property. This is primarily useful for debugging purposes.
- *testable*
  A read-only attribute which returns a boolean status code whether the function is testable. A *true* value means it has a scripted test function, and is of a suitable general function type.
- *testdata*
  The name of a file with sample structures and result data which can be used to verify the correct computation of property data in an installation. It should contain a reasonable set of structurally different structures and result data for this property which has been verified to be correct. The most suitable type of file for this data is a native **CACTVS** binary structure file.

When a property definition file is written with the `prop write` command, this data is encapsulated in the definition file. When the definition file is later read by an application, the data is written out to a temporary file, and the file name returned for this attribute is the name of the temp file, not the original file name. By this mechanism, it is not required to supply test data files as a separate files which may be difficult to locate and are easy to get lost.

If this field is empty, which is the default, no test data is present.

- *testfunction*
  The name of the test function of the property. If the property does not have a test function, the value is an empty string.

  A property test function is a parameterless TCL or PYTHON function which computes property data of the associated property on predefined structures, reactions, etc. and compares it to known results. In case of success, the function returns 1, 0 in case of failure, and it throws an error in case of a critical problem. If the property is a TCL script property, the test function is executed in the property slave interpreter. Otherwise, the global interpreter is used. If the property has a PYTHON computation function, the test function is also assumed to be PYTHON. Otherwise, it is expected to be written in TCL. Remote computation functions (synchronous or asynchronous server, mail service, etc.) currently cannot be tested.

- *timeout*
  A timeout for the property computation in seconds. If it is set to zero, the default, no timeout is in effect. Once a property computation exceeds its maximum computation time, it is aborted, and the computation fails. The timeout value is the total CPU time spent in the call to the computation routine. This includes all time spent with the computation of low-level properties which are requested inside the computation function of the current property.

  Computation aborts can lead to serious memory leaks. Time-outs should therefore not be used without need. Additionally, checking the time and setting interrupt handlers involves multiple system calls. The computation of trivial properties can slow down significantly if time-outs are enabled.

  Time-outs may be stacked if required, i.e. a timed computation routine can request the computation of another timed property. In that case, in the inner computation routine, both timers are active, and the timeout of any of these will lead to an overall computation failure.

- *trace*
  This is a deprecated attribute to control the tracing of computations in script functions. Please use the *flags* attribute.

- *traits*
  A couple of indicator bits describing the general type of the property. The value can currently be *none*, or any combination of *hashcode*, *stereodescriptor*, *rendering*, *pixelimage*, *vectorimage*, *linenotation*, *3dmodel*, *spectrum* and *interactive*.

- *unit*
  An information string describing the unit the proper data is using. The string is not parsed to enforce a recognized set of units, but built-in unit conversions only work if a standard unit is used. Supported unit sets currently include length, energy, mass, time, pressure, temperature, volume and mols. For any of these, a standard range of units is recognized (e.g. picoseconds/*ps* to years/*y* for time, and Joules, *kcals*, etc. for energy).

- *version*
  The version of the property definition. This is a string in a 1.2.3 (or shortened) style.

- *versionuuid*
  The version **UUID** associated with this property definition version.

- *warmupobjects*
  A list of decodable line notations (typically **SMILES**, Reaction **SMILES**, or **CACTVS** serialized objects) for use with the `prop use` command. The type of the object must correspond to the major object class associated with the property (i.e. an ensemble for ensemble/mol/atom properties, or a reaction for reaction properties). If you wish to specify the object class explicitly, alias attribute names *warmupstructures* or *warmupreactions* can also be used. By default, properties do not have associated warm-up objects. If `prop use` is invoked for properties without explicit warm-up objects, the default object in `::cactvs(default_warmup_structure)` or `::cactvs(default_warmup_reaction)` is used.

- *width*
  The width of the property. This is the number of slots the property is occupying in the data storage area of chemistry objects. The vast majority of properties has a width of one, and this is the default. The only standard exception are certain bond properties which are directional, i.e. there are two values, one for looking from the first atom to the second, and a different value for looking into the reverse direction. These bond properties have a width of two. In principle, it is possible to define properties of larger width, or to use this functionality on properties which are not attached to bonds.

  Property values of properties which are of a width larger than one are returned and input as lists in **TCL** scripts, or sequences in **PYTHON**. The length of these lists must be the same as the width in the property definition. It is not possible to access single-slot subsets of the property data directly.

  Use of this feature is no longer recommended. For multi-value properties, the use of *compound* data type properties, or a split into two independent properties are often better solutions.

- *windfall*
  A set of properties that is attached as result data as a side effect of computing this property. For local computations, this is for information only. For server property computations, it prepares the client to expect these additional property data records to be transmitted. Note that this list is not supposed to include minor, dependent properties which are needed as input for the computation. Rather, this should be used if there are two or more potentially interesting result properties which are most conveniently computed in parallel, and where computation of any of the additional properties in a separate request would essentially require to run the same computation over again.

> In a typical set-up, the windfall properties do not define their own computation functions, but redirect to the computation of this property via an *indirect* attribute, resulting in a set of two or more properties which are always computed together and where the combined computation function is maintained in a single place, on the property which lists them as windfall results.

The **`prop set`** command supports a special attribute value syntax for manipulating bitset-type attributes. If the first character of the argument is a minus character (-), the named bits in the set identified by the remainder of the argument are unset. If it is a plus (+), they are additionally set. With an equal sign (=), or no special lead character, the flag set replaces the old value. A leading caret character (^) toggles the selected bits.

Example:

```
prop set E_SCREEN flags +locked:fixedlength
prop set E_SCREEN flags -locked
```

prop setparameterdict

```
prop setparameterdict propertyname parametername datadict
Prop.Setparameterdict(property/pref,parametername,dict)
p.setparamdict(parametername,dict)
```

Modify an existing parameter of a property definition. The recognized dictionary items are the same as in the **`prop getparameterdict`** command, with the exception that all parsed values are ignored - parameter values, defaults, min and max values are defined via string data, for which a parse is attempted and an error generated if it fails. All dictionary items are optional. If a parameter definition attribute is not defined in the dictionary, the existing definition is retained.

If the named parameter is not already defined, an error is produced.

`prop setparamdict` is an alias to this command.

prop setparameter

```
prop setparameter propertyname ?parameter value?...
prop setparameter propertyname dictionary
p.setparameter(?parameter,value?,...)
p.setparameter(dict)
Prop.Setparameter(propertyname/pref,?parameter,value?,...)
Prop.Setparameter(propertyname/pref,dict)
```

Set zero or more computation parameters. In multi-threaded scripts, this effects only the thread-local parameter set. Only the base interpreter, and its slaves, manipulate the global parameter set, in addition to its own thread-local set.

In the second command variant, the dictionary argument must be a properly formed **Tcl** dictionary with suitable keyword/value pairs. If that is the case, the dictionary is implicitly expanded and processing proceeds as in the first command variant.

An attempt to set a parameter whose name is not listed in the current parameter set fails with an error. This is a safety mechanism to prevent typos in parameter names. Nevertheless, setting such an unknown parameter does succeed internally. In order to force the addition of a new parameter, the statement can be written with a **`catch`** command:

```
catch {prop setparam $pname my_new_parameter "New World!"}
```

This command can be abbreviated as **`prop setparam`**.

There is no command to unset a single parameter. This can be done by retrieving and manipulating the complete parameter list via a command like

```
set d [dict create {*}[prop get $propname parameters]]
dict unset d $obsolete_param
prop set $propname parameters $d
```

prop sqldecode

```
prop sqldecode propertyname value
Prop.Sqldecode(propertyname/pref,value)
p.sqldecode(value)
```

Attempt to decode a property value from a string formatted in **SQL**-style encoding. If it succeeds, its standard **Tcl** or **PYTHON** representation is returned. Otherwise, an error results.

prop sqlencode

```
prop sqlencode propertyname value
Prop.Sqlencode(propertyname/pref,value)
p.sqlencode(value)
```

Encode a property value in **SQL** formatting. The value parameter must be a valid representation of the property value in its **TCL** or **PYTHON** format. If the decoding succeeds, and the data type handler supports **SQL** formatting, the returned value is suitable for constructing **SQL** statements.

`prop sqlformat` is an alias for this command.

prop string

```
prop string propertyname
Prop.String(propertyname/pref)
p.string()
```

Get the **XML**-style property definition record as a string. This is equivalent to writing out the property definition and reading the output file as text data.

prop subcommands

```
prop subcommands
dir(Prop)
```

This command returns a list of all the subcommands of the **`prop`** command.

prop test

```
prop test ?propertyname?...
Prop.Test(?propertyname/pref?,...)
p.test()
```

Run the property test functions on a list of properties. The return value is a list of integers containing one integer per property. In case a listed property does not have a test function, the value is -1. If a test succeeds, the value is one. If it fails, the value is 0, and in case of a critical error in any of the tests, the test is canceled and the error reported.

prop unalias

```
prop unalias ?alias_name?...
Prop.Unalias(?alias_name?,...)
```

Remove one or more alias definitions introduced by a **`prop alias`** command.

If a specified alias name does not exist, it is silently ignored. If an existing property was hidden by an alias definition, it becomes visible again. However, existing property data in the name of the old alias on chemical objects continues to refer to the property definitions it was alias-mapped to when it was created.

prop use

```
prop use propertyname
p.use()
Prop.Use(propertyname/pref)
```

Warm up property computation, make sure that the property definition and associated computation dynamic libraries are loaded and the property computation is internally initialized. The latter function works by re-computing the property either on specifically set structures, reactions, etc. (*warmupobjects* attribute), or on default ensemble/reaction/etc. objects from **`::cactvs(default_warmup_structure)`** or **`::cactvs(default_warmup_reaction)`**. The class of warm-up object corresponds to the major object of the property (i.e. ensembles for atom/bond/ensemble properties, reactions for reaction properties, etc.). The warm-up objects are created only temporarily for the duration of the command, and a failure to compute the property on a warm-up object is no error.

Calling the computation function, if it exists, during program initialization is intended to allow the computation routine to perform initialization tasks, such as decoding fragments or setting global variables, so that later invocations in a multi-threaded script or application are safe even if the function writer was not careful about asset mutex locking during first-call initializations, and property database and property reference locking can be omitted if the overall property database is locked - see **`::cactvs(property_lock)`** control variable.

The command returns nothing.

prop verify

```
prop verify propertyname ?objecthandle?...
Prop.Verify(propertyname/pref,?objecthandle/objectref?,...)
p.verify(?objecthandle/objectref?,...)
```

Verify the syntactic correctness of property data on a list of major objects, such as ensembles, reactions or datasets. The return value is a boolean list of check results, 1 if the data is formatted correctly, 0 if not. If the property is not set for the object, the returned status value is -1.

In case there is no check function (attribute *checkfunction*, see **`prop set`** command) defined, or it cannot be found, an error results. More details on the operation of the check function can be found in the **`prop set`** subcommand documentation.

Example:

```
prop verify E_CAS $eh
```

This command checks if property `E_CAS` is set for the ensemble, whether it is formatted correctly by calling the scripted **Tcl** check function defined for property `E_CAS`. For this property, the check function runs the **CAS** check digit algorithm on the data value. It catches encoding errors where the check digit does not match the value as re-computed from the other digits.

`prop check` is an alias for this command.

prop write

```
prop write propertyname ?filename/dirname/emptystring?
p.write(?filename/dirname/emptystring?)
Prop.Write(propertyname/pref,?filename/dirname/emptystring?)
```

Write the current definition of the property to a file. The file can be loaded into other script interpreters explicitly (see **`prop read`** command) or implicitly via the property definition auto-load mechanism. If no file name is given, the name of the file is automatically constructed from the property name in lower case and the suffix *.xpd* for the new-style **`XML`** property definitions.

In addition to normal file names, the magic names *stdout* and *stderr* may be used, as well as already opened **Tcl** or **Python** socket and file handles, plus pipes indicated by a file name which starts with "|". Writing to **Tcl** or **Python** channels is not supported on the MS Windows platform. If the file name argument is a directory, the file name is still automatically generated, and the file written into that directory. If a file name with an explicit *.prp* suffix is specified, the output is written in the legacy keyword/value **ASCII**-based file format. Old property definition files continue to be supported for both input and output. They are backward-compatible with earlier toolkit versions, though they cannot store all content of the newer **XML** format.

If the output is written to a regular file named in the command, the suffix *.xpd* is automatically added if necessary, and the output contains a single property definition record. By writing to a **Tcl** or **Python** channel, it is possible to write multiple properties into a single file, and these can be read in a single operation by the **`prop read`** command. For explicit file input, neither one of the two standard suffixes nor the standard file name derived from the property name are mandatory, but the naming convention must be adhered to in order to enable auto-loading of property definitions.

If the file name argument is an empty string, or **`None`** for **Python**, the definition is returned as an in-memory string blob and not written to disk.

It is possible to write out built-in property definitions. For reference purposes, current toolkit distributions contain a **`BUILTIN`** directory which contains dumps of the property definitions of the built-in property set.

The command returns the name of the file written to if the output is not a blob.

prop xcreate

```
prop xcreate propertyname ?attribute value?...
prop xcreate propertyname dict
Prop.Xcreate(propertyname,?attribute,value?,...)
Prop.Xcreate(propertyname,dict)
Prop.Xcreate(propertyname,?attribute=value?,...)
```

Create a new property definition, and optionally set attributes.

The command, and the difference to the normal `prop create` command is explained in the paragraph on that command.

The repx Command

Intentionally undocumented, internal use only

The report Command

The **`report`** command is used to interact with report objects. The purpose of report objects is to generate reports in **PDF** or **HTML** format from structure data. **PDF** files can contain a hidden structure-searchable database with the structure and rendered or auxiliary data items.

The format of a report is defined by an **XML** style file, which describes a record box with arbitrary positioning of data or image fields. These record boxes can be stacked horizontally and vertically on report pages. The **XML** style file is embedded in **PDF**s written by report objects, so these files can contain both human-readable and computer-readable data content plus the layout information used to generate the file. It is for example possible to re-render content with a different style file, or use the style file extracted from a template file to render additional documents in exactly the same style.

The unit for describing report object placements are 72 DPI pixels. The are converted and/or scaled to the selected output format.

report create

```
report create ?attribute value?...
report create dict
Report(?attribute,value?,...)
Report(dict)
Report(attribute=value,...)
Report.Create(?attribute,value?,...)
Report.Create(dict)
Report.Create(attribute=value,...)
```

Create a new report object. Additionally, an set of attributes of the object may be defined in the same statement. The settable attributes are the same as in the **`report set`** command.

The command returns the handle or reference of the new report object.

report delete

```
report delete ?rhandle?...
report delete all
r.delete()
Report.Delete(?rhandle/rref?,...)
Report.Delete("all")
```

Delete one of more report objects which are identified by their handles or references. The second variant deletes all currently defined report objects.

Example:

```
report delete {*}[report list]
```

This command is equivalent to **`report delete all`**.

report extract

```
report extract rhandle pdf_filename ?mode? ?outfilenamenbase?
r.extract(filename=,?mode=?,?outfilenamebase=?)
```

Extract the **SQLITE** database and the report style **XML** file from a **PDF** report.

The default mode is all, meaning that both the database and the formatting information is extracted, written to external files, and attached to the report object. Other supported modes are *style* (formatting only) and *database* (data only).

The final optional argument sets the base filename of the extracted **SQLITE** and **XML** style files. If it is not set, a file name is automatically generated.

The return value of the command depends on the extraction mode. In the *all* mode, is is a list of the respective filenames for the database and style file. In other modes, it is a single string with the specific filename.

report get

```
report get rhandle attribute
r.get(attribute)
r.attribute
r[attribute]
```

Query an attribute of a report object. The currently supported attributes are:

- *address_city*
  The city part of the author contact address.
- *address_country*
  The country part of the author contact address, following the ISO3166 standard.
- *address_state*
  The state part of the author contact address. Empty if not applicable.
- *address_street*
  The street address part of the author contact address. Includes floor, house number, etc.
- *address_zip*
  The **ZIP** code or other applicable postal code of the author contact address.
- *affiliation*
  The institution the author of the report works for.
- *affiliationduns*
  The **DUNS** registration ID of the affiliated institution. This is primarily useful for US government projects.
- *affiliationurl*
  The **URL** of the affiliated institution.
- *author*
  The author of the report, as free-form text.
- *authorurl*
  A **URL** with information on the author, or an empty string if unset.
- *border*
  The distance between the paper or **HTML** page and the first elements of a data box.
- *boundingbox*
  A list of the upper left x/y and lower right x/y coordinates of the current layout, with all scaling and offset computations applied. This is a read-only attribute.

- *category*
  A category string to be used if the report definition is stored in a repository.
- *classuuid*
  The base class **UUID** of this report.
- *comment*
  A free-form comment.
- *date*
  The date of the last change of the report.
- *dbfields*
  A nested list of the columns defined for the embedded structure and reaction database. Each field is reported as a list of the property or pseudo-property name of the field, the true database column name, a bitset set of any database field attributes set on the field (from the bit set of *none*, *index*, *unique*, *notnull*, *nocompute*), the name of the **SQLITE** database type used to encode the field, and the field length, which is zero if it is not defined.

  If database fields have not yet been defined on the report object, querying or setting this attribute instantiates a standard set for a structure (but not reaction) database. Additionally, all properties currently associated with a data display field in the layout are automatically added, so that every visible data value can also be read from the database.

  When setting this attribute, every field sublist must contain only between one and three elements. The last two items reported are automatically deduced from the type of data to be stored. If no explicit field name is specified, the field name is copied from the first argument. If no third element is supplied, the database column has no auxiliary **SQL** attributes, i.e. it is not indexed, may contain multiple entries of the same value, can be `NULL`, and an attempt will be made to compute its value when a structure or reaction record is written and the ensemble or reaction does not currently hold a value for the property or pseudo-property of the database column.
- *dbfilename*
  The name of the **SQLITE** database file with the structure- and data-searchable content associated with this report object. It can also be set, though usually its name is auto-generated. Initially, an empty return value indicates that no file name has been set.
- *doi*
  A digital object identifier for the report, if defined.
- *email*
  An email contact address of the author.
- *eolchars*
  The end-of-line character to use in **ASCII**-formatted output, such as the **HTML** report style. The default value is the platform style (Windows, Mac or Unix/Linux). It can be set to any string. Additionally, the magic values *win* (or *crlf*), mac (or *cr*) and *linux* (or *unix*, or *lf*) are recognized.
- *footer*
  The center-aligned page footer text, which can contain placeholders. The default is an empty string.

- *handle*
  The **TCL** handle in string format
- *header*
  The center-aligned page header text, which can contain placeholders. The default is an empty string.
- *htmlfilename*
  The name of the file with the **HTML**-formatted rendering of the report content. Usually, this is automatically set. The initial value is an empty string, which results in an automatically generated name if a **HTML** file is written.
- *infourl*
  A **URL** with information on the report, or an empty string if unset.
- *keywords*
  A list of keywords associated with the report
- *leftfooter*
  The left-aligned page footer text, which can contain placeholders. The default is an empty string.
- *leftheader*
  The left-aligned page header text, which can contain placeholders. The default is an empty string.
- *versionuuid*
  The instance **UUID** associated with this report
- *license*
  The license class associated with this report. Setting the license to a standard type updates the associated **URL** with a standard location.
- *licenseurl*
  A **URL** with details about the report license.
- *literature*
  This is a free-form string for a literature reference describing the meaning and/or algorithm behind the report.
- *maxpages*
  The maximum number of pages to write, in **PDF** or **HTML** format. If the current data set is too large to fit into this number of pages, the rest is silently omitted. A negative value, which is the default, allows an unlimited number of pages. In any case, the maximum page count of a **PDF** report is 9999 pages. This is due to a limitation in the **PDF** rendering library.
- *maxrecs*
  The maximum number of records to write, which is the same as the number of top-level data boxes. The actual size of the dataset or structure file used as data source for the report may be larger, if a filtering mechanism, such as selection processing, is set. Records in excess of the maximum are silently omitted. A negative value, which is the default, allows an unlimited number of records.

- *name*
  The primary name of the report.
- *orcid*
  The **ORCID** code of the author of the report (see www.orcid.org).
- *orientation*
  The page orientation, which can either be *portrait* (the default) or *landscape*.
- *ownerpassword*
  The owner password for generated **PDF** files. An empty string, which is the default, indicates that no password is set. For **HTML** output, this attribute is ignored.
- *paper*
  The size of paper, such as *A4*, *A3* or *letter*, which is used to format the output as **PDF** or **HTML**. This parameter has an effect even on **HTML** output, since the code contains page breaks for nice printing. The default is the toolkit default paper, which itself is usually A4 or letter.
- *path*
  The repository path for displaying hierarchical repository trees. This attribute is independent of any file system paths.
- *pdffilename*
  The name of the file with the **PDF**-formatted rendering of the report content. Usually, this is automatically set. The initial value is an empty string, which results in an automatically generated name if a **PDF** file is written.
- *phone*
  A contact phone number of the author.
- *references*
  Cross references of the report. This is a nested list of class **UUID**s and reference type tags.
- *regid*
  For registered reports, the registration ID. Unregistered modules report zero.
- *rightfooter*
  The right-aligned page footer text, which can contain placeholders. The default is an empty string.
- *rightheader*
  The right-aligned page header text, which can contain placeholders. The default is an empty string.
- *scale*
  A scaling factor to convert the 72-DPI internal coordinates to print coordinates. The default scale is 1.0. If set, the magic value *#auto* is also recognized, which sets the scaling factor so that the configured number of horizontal and vertical blocks with the current style file just fit onto the configured paper type. Please refer also to the interacting *xblocks* and *yblocks* attributes. Automatic setting of this attribute is only possible if a style file has been read, because it requires the box layout data.

- *spacing*
  The number of pixels between top-level data boxes if more than one is stacked in x or y direction. The default spacing is 1.
- *styledata*
  The contents of the current style file, in original **XML** formatting. This data can be set either directly as a string, or indirectly via the *stylefilename* attribute. If set directly, the *stylefilename* attribute is reset.
- *stylefilename*
  The name of the **XML** file with report render style information. Its initial value is an empty string. If this attribute is set, the file contents are parsed and stored internally. The file contest are then available as in the *styledata* attribute.
- *title*
  The title of the report as a free-form string. In **HTML** reports, it is used as page title. In **PDF**, it is part of the document metadata.
- *userpassword*
  The user password for **PDF** documents. By default, it is an empty string, which means that no password is set. For **HTML** output, the attribute is ignored.
- *version*
  A version string
- *versionuuid*
  The version **UUID** associated with this report version.
- *xblocks*
  The number of top-level data boxes to arrange in horizontal direction on a document page. When it is set, the magic words *#auto* is recognized, which computes the maximum number of boxes which can be placed without overspill on the page using the current border, spacing and scaled top-level box size. Please refer also to the interacting *scale* attribute. Automatic setting of this attribute is only possible if a style file has been read, because it requires the box layout data.
- *yblocks*
  The number of top-level data boxes to arrange in vertical direction on a document page. When it is set, the magic words *#auto* is recognized, which computes the maximum number of boxes which can be placed without overspill on the page using the current border, spacing and scaled top-level box size. Please refer also to the interacting *scale* attribute. Automatic setting of this attribute is only possible if a style file has been read, because it requires the box layout data.

report list

```
report list ?pattern?
Report.List(?pattern=?)
```

Return the handles of all currently defined report objects. The handles may optionally be filtered by a standard **Tcl** string match condition.

report read

```
report read rhandle pdf/db/html_filename ?dataset?
r.read(filename=,?target=?)
```

Read data for a report from a **PDF** file with an embedded database, a **SQLITE** database with the property structure (typically extracted from a report **PDF**) or a **HTML** file generated by the report command with embedded data.

If an existing destination dataset is specified, all read structure or reaction objects are put into that dataset. A new dataset is automatically created if the argument is omitted, specified as an empty string (including **None** for **PYTHON**), or as the magic name *#new*.

The return value is the handle or reference of the dataset holding the extracted chemistry objects.

report ref

```
Report.Ref(identifier)
```

PYTHON-only method to get a reference of the report object from its handle.

report set

```
report set rhandle ?attribute value?...
report set rhandle dict
r.set(?attribute,value?,...)
r.set(dict)
r.attribute = value
r[attribute] = value
```

Set attributes of the report object. The attributes are explained in the section on the **report get** command.

Example:

```
report set $rhandle stylefile „mystyle.xml“ paper A4 xblocks 2 yblocks #auto \
   scale #auto
```

This is a typical page set-up operation. The PDF or HTML form of the report will contain 2 top-level boxes arranged horizontally, a suitable (as computed from the aspect ratio of the paper) number of vertically stacked blocks, and everything scaled so that either the horizontal or vertical extent of the block stacks, plus the border, just fit onto the selected paper, without overspill.

report subcommands

```
report subcommands
dir(Report)
```

Return a list of all subcommands of report objects. The command does not have any parameters.

report write

```
report write rhandle ?filename? ?source? ?flagdict?
report write rhandle ?filename? ?source? ?flagname flagvalue?...
r.write(?filename=?,?source=?,?flags=?)
```

Write a report file from the existing report definition and a structure data source.

The file name specifies an output data file. Its type is initially inferred by the filename suffix - *.pdf* for **PDF** files, *.htm* or *.html* for **HTML** files, and *.db* or *.sqlite* for a raw **SQLITE** database. The file name can be omitted if the *pdffilename* attribute is set, and the output format should be **PDF**, or another format is set in the *flags* argument and the corresponding format-specific filename attribute is set. If an explicit format is specified in the *flags* argument, the automatic format selection from the filename is overridden. The filename can also be a directory. In that case, an automatically generated file name based on the report instance **UUID** is constructed and the file written to that directory. The magic output file names *stdout* and *stderr* are also recognized.

The structure data source can either be the handle or reference of a dataset with structure or reaction objects, or the handle (reference) of an open *molfile* structure or reaction file. If no structure object data source is specified, but there is a current **SQLITE** database file associated with the report object, data is read from that database.

The output flags can be from the following set:

- *format* The desired output format. It can be *pdf*, *html* (*htm*) or *db* (*sqlite*).
- *maxrecs* The maximum number of structure objects or structure input file records to process.
- *page* A page or page range. Pages outside the range for which data exists are omitted.
- *reusedb* A boolean flag indicating whether an existing **SQLITE** database file should be re-used.
- *selection* The method to pick structures or reactions for inclusion in the report. It can be *ignore* (all dataset items or file records are reported), *filter* (only items passing a filter are included in the report) or *highlight* (only items where the *highlight* attribute is set are included).

The return value of the command is the output file name, which may have been auto-generated.

The soap Command

The **soap** command is used to facilitate networked communication by means of **SOAP** messages. **SOAP** message objects are also useful to parse other styles of **XML**-formatted messages which are not really **SOAP**, for example **NCBI** Entrez *eutils* result messages.

SOAP objects are created like other **CACTVS** objects, and have internal state, so it is possible to communicate with multiple sources simultaneously by creating a **SOAP** object for every channel.

SOAP objects share many characteristics with **JSON** objects, and many commands are identical or at least very similar.

These are the currently supported **SOAP** object commands:

soap append

```
soap append soaphandle ?attribute value?...
soap append soaphandle dict
s.append(?attribute,value?,...)
s.append(?attribute=value?,...)
s.append(dict)
```

This is a variant of the **soap set** command. The difference is that the supplied data is appended to the current attribute value instead of replacing it. In case appending is not a possible operation, the result is the same as using **soap set**.

The set of supported attributes is explained in the paragraph on **soap set**.

The command returns the original object handle or reference.

soap create

```
soap create ?attribute value?...
soap create dict
Soap(?attribute,value?,...)
Soap(dict)
Soap(?attribute=value?,...)
Soap.Create(?attribute,value?,...)
Soap.Create(dict)
Soap.Create(?attribute=value?,...)
```

Create a new **SOAP** object. The return value is the new object handle or reference. If no extra attributes are specified, an empty object with default settings is created. Processing of specified optional attribute/value pairs is performed in as an identical fashion to the **soap set** command.

soap delete

```
soap delete ?soaphandle?...
soap delete all
s.delete()
Soap.Delete(?sref/shandle?,...)
Soap.Delete(“all”)
```

Destroy one or more **SOAP** objects. The special word *all* can be used to remove all **SOAP** objects currently defined in the interpreter.

For the sake of consistency with the commands for other objects, **`soap close`** is an alias to this command.

The return value is the number of successfully deleted **SOAP** objects.

soap error

```
soap error soaphandle ??message? channel? ?code? ?factor?
s.error(?message=?,?channel=?,?code=?,?uri=?)
```

Send a properly formatted **SOAP** error message over a **Tcl** channel to a client, or at least prepare it. If any of the optional arguments behind the channel argument are provided, they are processed like equivalent **`soap set`** commands and change the internal attributes of the object.

In the generated message, the *message, code* and *factor* values are wrapped into **XML** tags of type *faultstring*, *faultcode* and *faultfactor*, respectively. Usually, the *factor* value should be set to the **URI** of the service, which could be stored in the *uri* attribute of the **SOAP** object. This core information is then embedded into a standard **SOAP** fault envelope. If no channel argument or an empty string is specified, and the internal channel of the **SOAP** object is undefined, no data is transmitted over the channel, but the message is still formatted. If a valid channel was specified, or the **SOAP** object has a valid internal channel, the message is immediately transmitted.

The return value of the command is the complete **SOAP** error message, with header and body. The internal header and body fields of the **SOAP** object are also updated and allow subsequent individual retrieval of these components.

soap find

```
soap find soaphandle taglist ?mode? ?contentflags?
s.find(tags=,?mode=?,?contentflags=?)
```

Find the contents and/or attributes of specific **XML** tags in a message. The generation of a parse tree (see **`soap parse`** command) is a prerequisite for this function. If a parse tree does not yet exist, but the **SOAP** object has body data, an attempt is automatically made to execute a parse of the full message body data.

The command returns data found inside the first of potentially multiple query tags which matches in case-sensitive fashion a tag in the parse tree, which is traversed depth-first.

The *mode* argument determines the reporting mode. It can be one of

- *first*
  Find the first instance of tag and return single data item.
- *next*
  Find the next instance of tag after the current position, or the begin of the message if this is the first query, and return single data item.
- a numerical index
  Find the *nth* instance of the tag and return single data item. The index starts with zero.
- *all*
  Return all matching tag contents as a list of data item information. This is the default mode.

The type of tag content reported can be configured by the *contentflags* argument. The argument can be an arbitrary list made from the words

- *tag*
  The name of the matched tag itself.
- *attributes*
  The attributes of the tag, as name/value dictionary.
- *chardata*
  The character data in the tag, excluding embedded inner tags, possibly trimmed (see *trimmed* flag word).
- *children*
  If set, include the same data as for the matched tag recursively for all its children, as nested lists.
- *trimmed*
  Remove left and right white space from the returned *chardata* value.
- *noleafchildren*
  If *children* is set, omit output of an empty children list if a node has no children.

The default output selection is just *chardata*. If multiple content flags are selected, the order of the elements in an item output list is *tag*, *attributes*, *chardata* and finally *children*. If a query fails to yield any results, an empty string is returned. In reporting mode *all*, the item output lists are themselves list elements of the overall result list.

This command is not **SOAP**-specific but can work in principle on any **XML** data, for example **Entrez** *eutils* output.

soap get

```
soap get soaphandle attribute
s.get(attribute)
s.attribute
s[attribute]
```

Query the value of an attribute of a **SOAP** object. The list of recognized attributes is explained in the paragraph on the **`soap set`** command.

The return value of the command is the value of the attribute.

soap list

```
soap list ?pattern?
Soap.List(?pattern=?)
```

Return a list of the handles or references of all currently existing **SOAP** objects in the interpreter. If desired, the list can be filtered by a string pattern.

soap parse

```
soap parse soaphandle ?starttag? ?classname/instanceindex? ?data?
s.parse(?starttag=?,?class=?,?data=?)
```

Parse an **XML** message stored in the *body* attribute of the object and create the parse tree which is the prerequisite for retrieval of message content via the **`soap find`** command. If a *data* argument is supplied, it replaces the *body* data of the object.

By default, or when the start tag argument is set to an empty string (or **`None`** for **PYTHON**), the complete message is parsed. Optionally, processing can be limited to a subsection of the full data that starts with the specified tag. In that case, only the content between the selected opening and closing tags matching the supplied start tag is analyzed, and the argument replaces the *tag* attribute of the **SOAP** object. By default the first start tag match is used, but this can be changed with the next optional argument.

In case of multiple tags with the same name in the parsed data, an additional class attribute can be specified as a second element. If this optional filtering argument is used, the first tag block matching both the tag name and having a matching *class* attribute is extracted and parsed. If the class argument is omitted, or empty, no class filtering is performed. Instead of a class name, it is also possible to specify an integer instance index in the same argument position. By default, the first occurrence of the start tag (index 0) is parsed, but by specifying a different start tag index, a different tag instance may be selected.

The result of the command is the parse tree in nested list representation. This is the same data as the *parsetree* attribute which can be queried via **`soap get`**.

soap read

```
soap read soaphandle ?channel?
s.read(?channel=?)
```

Read a message with a **HTTP** header from a channel. If the channel argument is not supplied, the internally configured **SOAP** object channel is used. The internal header and body attributes of the object are updated when the read succeeds. Note that this command works with any communication which uses a **HTTP** header and body data. It is not limited to reading **SOAP** messages, or even plain **XML** data.

The reader supports chunked data transmission.

The return value of the command is the received body text. In most applications, this command is followed by a **`soap parse`** command, and then multiple **`soap find`** commands.

soap ref

```
Soap.Ref(identifier)
```

PYTHON-only method to get a reference of the **SOAP** object from its handle.

soap reformat

```
soap reformat soaphandle ?text?
s.reformat(?text=?)
```

Pretty-print an **XML** blob, with properly indented nested tags for easy visual inspection. If no explicit string is specified, the current *body* attribute value of the soap object is used. The return value of the command is the reformatted content. The original data is not changed.

This command assumes that the input is not pre-formatted. It does not attempt to remove existing line feeds for formatting whitespace on the input, but rather adds it own formatting on top.

soap request

```
soap request soaphandle ?method? ?channel? ?parameters? ?uri?
s.request(?method=?,?channel=?,?parameters=?,?uri=?)
```

Call a **SOAP** handler on a remote server, or at least prepare the message content to do that. If any of the optional arguments with the exception of the *channel* argument are provided, they are processed like **`soap set`** commands and change the internal attributes of the object. If the *uri* argument is not supplied, and the **SOAP** object has no configured *uri* attribute, a default **URI** to be used as part of the **HTTP POST** instruction in the header is constructed from the *host*, *port* and *method* attributes. This **URI** is not directly used to open a communication channel, though - the **TCL** transmission channel can have been opened by any suitable means.

If the channel argument is not supplied, or written as an empty string, and the object has no valid internal handle, the message is just assembled. Otherwise, it is transmitted directly. Using an explicit channel argument does not update the internal channel attribute of the object.

The return value of the command is the assembled **SOAP** request message, with header and body. The internal header and body fields of the **SOAP** object are also updated and allow subsequent individual retrievals of these components.

Note that this command just sends the request, but does not wait for or read the server response. Use the **`soap read`** command for this purpose.

soap respond

```
soap respond soaphandle responsedata ?channel? ?method? ?namespace?
s.respond(data=,?channel=?,?method=?,?namespace=?)
```

Send a properly formatted **SOAP** response message over a **TCL** channel to a client, usually as a response after receiving a **SOAP** request message from that client. If any of the optional arguments behind *channel* are provided, they are processed like **`soap set`** commands and change the internal attributes of the object

In the generated message, the *responsedata* value is wrapped into an **XML** tag of type <*methodResponse*>, where *method* is replaced by the name of the configured **SOAP** method. This core information is then embedded into a standard **SOAP** envelope and transmitted over the channel. If no channel argument is given, and the internal channel of the **SOAP** object is undefined, the message is just formatted, but not transmitted.

The return value of the command is the complete **SOAP** message, with header and body. The internal *header* and *body* fields of the **SOAP** object are also updated and allow subsequent individual retrievals of these components.

soap set

```
soap set soaphandle ?attribute value?...
soap set soaphandle dict
s.set(?attribute,value?,...)
s.set(dict)
s.set(attribute=value,...)
s.attribute = value
s[attribute] = value
```

Set one or more attributes of a **SOAP** object. Since this paragraph is also referenced from the `soap get` subcommand, the attribute set listed here includes attributes which cannot be set, or for which setting them to any scripted value does not usually make sense.

The currently supported set of attributes is:

- *body*
  The body part of the last received **SOAP** message. This attribute is not usually set by scripts.
- *bodylength*
  The length of the body, in bytes. This is a read-only attribute.
- *channel*
  The communication channel associated with the **SOAP** object. This is a standard **Tcl** channel handle. It is possible to set it to an empty string (or `None` in **Python**), which indicates that no channel is set.
- *errorcode*
  The error code from the last **XML** parsing step as integer value. If no error occurred, the value is 0.
- *errorcolumn*
  The column the **XML** parser detected the last error.
- *errorfactor*
  The factor leading to an error, either as received from a **SOAP** message, or set in preparation of sending an error message. Standard practice is to send the **URI** of the failed service in this field.
- *errorline*
  The line the **XML** parser detected the last error.
- *errormsg*
  The message of a **SOAP** error, either as extracted from a **SOAP** message, or set in preparation of sending an error message.
- *header*
  The header part of the last received **SOAP** message. This attribute is not usually set by scripts.
- *headerlength*
  The length of the header, in bytes. This is a read-only attribute.
- *host*
  The remote host the **SOAP** object communicates with.
- *method*
  The method identification of either the last received **SOAP** message, or set in preparation to send a message.

- *namespace*
  The **SOAP** method namespace. It can be set to an empty string in order to avoid using method namespace attributes in sent **XML** messages altogether. The namespace suppression only applies to the method section. The overall **SOAP** envelope will still contain a standard set of namespace references.

- *parameters*
  The **SOAP** method parameters. If queried, this is a dictionary of name/value pairs. In order to set these, a properly formatted dictionary must be supplied. The formatting is the same as for the *parameters* attribute of property definitions.

- *parsetree*
  A nested list representation of parse tree generated by the last successful **XML** content parse by means of the **`soap parse`** command. This is a read-only attribute, and primarily useful for debugging.

- *port*
  The port the **SOAP** object uses for network communication. The default is 80, the standard **HTTP** port.

- *response*
  The response part of the last **SOAP** message, in parse tree representation. This is a read-only attribute.

- *result*
  The pure method invocation result data extracted from the last **SOAP** message. This is a read-only attribute.

- *tag*
  The current start tag for parsing a subsection of **XML** message content, see **`soap parse`** command.

- *uri*
  The URI associated with a **SOAP** service.

soap subcommands

```
soap subcommands
dir(Soap)
```

This command returns a list of all the subcommands of the **`soap`** command.

The tablex Command

The **`tablex`** command is used to manage table file format handler extensions. The command has the following subcommands:

tablex defined

```
tablex defined format
Tablex.Defined(format)
```

A boolean check whether a specific table file format is supported by an I/O handler. If the format is not yet known, an attempt is made to locate and auto-load the I/O module. For an equivalent command without auto-loading, see **`tablex exists`**.

tablex exists

```
tablex exists format
tx.exists()
Tablex.Exists(format)
```

A boolean check whether a specific table file format is supported by an I/O handler. No attempt is made to auto-load a handler module if it is not already in memory. The name can be the primary name of the table file format, or any recognized alias. For an equivalent command with auto-loading, see **`tablex defined`**.

tablex get

```
tablex get format attribute
Tablex.Get(format,attribute)
tx.get(attribute)
tx.attribute
tx[attribute]
```

Query the value of an attribute of the table file format handler. The list of supported attributes is described in the section on the **`tablex set`** command.

If the format is not yet known, an attempt is made to auto-load it.

tablex list

```
tablex list ?pattern?
Tablex.List(?pattern=?)
```

Return a list of all currently supported table formats, including those handled by built-in format handlers. If desired, the list can be filtered by a string pattern.

tablex load

```
tablex load format ?objectfile?
tablex load all
Tablex.Load(format,?objectfile?)
Tablex.Load("all")
```

Explicitly load a table file format handler module. If the module is already loaded, the current version is unloaded first. If no specific object file (a shared library on Unix/Linux, a DLL on Windows, a bundle file for MacOSX) is specified, the standard name of the module file is automatically generated from the data type name, and then the file searched in the directories in the data type handler module path. The module search path can be customized in the control variable **`::cactvs(tablexpath)`**.

For TCL, the return value of the command is the slot in the table file format module table the module has been loaded into. This corresponds to the value of the *slot* attribute which can be queried via **`tablex get`**. For PYTHON, the return value is a module reference.

The second form of the command scans the currently set table format extension search path and loads all accessible modules which are not yet in memory. Modules which are already active in the running application are not unloaded, and only a single instance of each I/O module, even if present under various alias names in the module directories, is loaded. This form of the command does not return a value.

tablex modules

```
tablex modules ?pattern?
Tilex.Modules(?pattern=?)
```

This is an alias for **`tablex list`**.

tablex ref

```
Tablex.Ref(format)
```

PYTHON-only method to get a reference of the module, which allows terser attribute retrieval commands and other operations.

tablex reload

```
tablex reload format ?objectfile?
Tablex.Reload(format,?objectfile?,...)
```

A variant of the **`tablex load`** command which fails if the I/O module was not previously loaded. There is no *all* variant of this command.

tablex subcommands

```
tablex subcommands
dir(Tablex)
```

Return a list of all supported subcommands of the **`tablex`** command in the current interpreter.

tablex set

```
tablex set format ?attribute value?...
tablex set format dict
Tablex.Set(format,?attribute,value?,...)
Tablex.Set(format,dict)
Tablex.Set(format,attribute=value,...)
tx.set(?attribute,value?,...)
tx.set(dict)
tx.set(attribute=value,...)
tx.attribute = value
tx[attribute] = value
```

Set one or more attributes of the table file format handler. Some attributes are read-only. They are still listed here because the tablex get command refers to this section. The following attributes are recognized:

- *address_city*
  The city part of the author contact address.
- *address_country*
  The country part of the author contact address, following the ISO3166 standard.
- *address_state*
  The state part of the author contact address. Empty if not applicable.
- *address_street*
  The street address part of the author contact address. Includes floor, house number, etc.
- *address_zip*
  The **ZIP** code or other applicable postal code of the author contact address.
- *affiliation*
  The institution the author of the table extension definition works for.
- *affiliationduns*
  The **DUNS** registration ID of the affiliated institution. This is primarily useful for US government projects.
- *affiliationurl*
  The **URL** of the affiliated institution.
- *aliases*
  A list of alternative names that are recognized as alternative names for the table file format.
- *author*
  The author of the module, as free-form text.
- *authorurl*
  A **URL** with information on the author, or an empty string if unset.
- *builtin*
  A boolean flag indicating whether the format handler is built-in. This is a read-only attribute.
- *category*
  A category string to be used if the table extension is stored in a repository.
- *charset*
  The character set the file format uses.
- *classuuid*
  The base class **UUID** of this module.
- *comment*
  A free-form comment.
- *date*
  The date of the last change of the module source code.

- *doi*
  A digital object identifier for the module, if defined.
- *email*
  An email contact address of the author.
- *features*
  A list of the types of functions the module provides in its function table. Function types include *check* (auto-identify file format), *read* (read that format) and *write* (write that format).
- *infourl*
  A URL with information on the module, or an empty string if unset.
- *keywords*
  A list of keywords associated with the module.
- *license*
  The license class associated with this module. Setting the license to a standard type updates the associated **URL** with a standard location.
- *licenseurl*
  A **URL** with details about the module license.
- *literature*
  This is a free-form string for a literature reference describing the meaning and/or algorithm behind the module.
- *mimetype*
  The **MIME** type of the table file format, for example *application/ms-excel.*
- *name*
  The primary name of the table file format. Because the format may have been specified via an alias name, this may not be the same as he command parameter.
- *objectfile*
  The full path name of the object file (shared library, DLL or bundle file) of the module. For built-in modules, this is an empty string.
- *orcid*
  The **ORCID** code of the author of the module (see www.orcid.org).
- *parameters*
  A keyword/value dictionary of format-specific I/O attributes with their default values which are not represented as general table object attributes. Upon file output with a specific format, parameters from this dictionary which have not been explicitly configured in the *parameters* attribute of the table object to be written are added from the table I/O module instance to it. Dictionary keys already set as a custom table *parameters* entry are not overwritten.
- *path*
  The repository path for displaying hierarchical repository trees. This attribute is independent of any file system paths.

- *phone*
  A contact phone number of the author.
- *references*
  Cross references of the module. This is a nested list of class **UUID**s and reference type tags.
- *regid*
  For registered modules, the registration ID. Unregistered modules report zero.
- *slot*
  The table file format handler table slot the module is loaded into. This attribute is read-only.
- *sourcefile*
  The source file of the module, if available.
- *suffixes*
  A list of file name suffixes associated with this format. This is a read-only attribute.
- *version*
  A version string.. This is a string in a 1.2.3 (or shortened) style.
- *versionuuid*
  The version **UUID** associated with this module version.

tablex unload

```
tablex unload ?format?...
tx.unload()
Tablex.Unload(?txref/format?,...)
```

Unload one or more table file format handler modules. It is an error to specify the name of a module which is not loaded. Built-in modules cannot be unloaded.

The return value is the number of unloaded modules.

The typex Command

The **`typex`** command is used to manage data type handler extensions. The command has the following subcommands:

typex defined

```
typex defined datatypename
Typex.Defined(datatypename)
```

A boolean check whether a specific data type is supported by a data type handler. If the format is not yet known, an attempt is made to locate and auto-load the handler module. For an equivalent command without auto-loading, see **`typex exists`**.

typex exists

```
typex exists datatypename
tx.exists()
Typex.Exists(datatypename)
```

A boolean check whether a specific data type is supported by a data type handler. No attempt is made to auto-load a handler module if it is not already in memory. The name can be the primary name of the data type, or any recognized alias. For an equivalent command with auto-loading, see **`typex defined`**.

typex get

```
typex get datatypename attribute
Typex.Get(datatypename,attribute)
tx.get(attribute)
tx.attribute
tx[attribute]
```

Query the value of an attribute of the data type handler. Note that data type handlers are static - it is neither possible to define them on the command line, nor to change any attribute. Therefore, there are no **`typex create`** or **`typex set`** commands.

The following attributes are recognized:

- *address_city*
  The city part of the author contact address.
- *address_country*
  The country part of the author contact address, following the ISO3166 standard.
- *address_state*
  The state part of the author contact address. Empty if not applicable.
- *address_street*
  The street address part of the author contact address. Includes floor, house number, etc.
- *address_zip*
  The **ZIP** code or other applicable postal code of the author contact address.
- *aliases*
  A list of recognized alias names of the data type.

- *affiliation*
  The institution the author works for.
- *affiliationduns*
  The **DUNS** registration ID of the affiliated institution. This is primarily useful for US government projects.
- *affiliationurl*
  The **URL** of the affiliated institution.
- *author*
  The author of the module, as free-format text.
- *authorurl*
  A **URL** with information on the author, or an empty string if unset.
- *builtin*
  A boolean flag indicating whether this is a built-in type.
- *category*
  A category string to be used if the module is stored in a repository.
- *classuuid*
  The base class **UUID** of this type extension
- *cmpops*
  A list of the comparison operation flags the handler supports. Please refer to the **`prop compare`** command for a list of possible flags.
- *comment*
  A free-form comment
- *date*
  The date the module source code was last changed.
- *doi*
  A digital object identifier for the module, if defined.
- *elementsize*
  In case the data type has vector elements or other components which can be addressed in a vector-like fashion, and the size of these components is constant, this attribute reports the size of an element. A zero size is reported if this attribute is not applicable.
- *elementtype*
  The data type of vector elements or other components which can be addressed in a vector-like fashion. For data types without sub-elements of a constant type, this is an empty string.
- *email*
  An email contact address of the developer of the module.
- *flags*
  A collection of flags reporting specific properties of the datatype. The currently supported set is:
  - *none*
    No flags

  - *indexable*
    The data type supports the extraction of components.
  - *knimecompatible*
    This data type has a compatible data type in the standard datatypes of the **KNIME** software.
  - *multiline*
    The data type produces multi-line file output in standard formatting
  - *needswritelock*
    Internal use only.
  - *nominmax*
    The data type has no reasonable definition, or use of, minimum and maximum values.
  - *precision*
    The data type supports the concept of numerical precision.
  - *simplenumeric*
    This is a simple numeric type.
  - *vector*
    The data type is a vector type with identical elements that can be accessed by a numeric index.
- *infourl*
  A **URL** with information on the module, or an empty string if unset.
- *keywords*
  A list of keywords associated with the module.
- *license*
  The license class associated with this module. Setting the license to a standard type updates the associated **URL** with a standard location.
- *licenseurl*
  A **URL** with details about the module license.
- *literature*
  A free-form literature reference.
- *indexing*
  A descriptive string with a short summary of the field access functions supported by this data type handler module.
- *isarray*
  A boolean flag indicating whether this data type is an array or not.
- *name*
  The primary name of the data type. Since the information may be queried via an alias name, this can be different from the command argument.
- *objectfile*
  The full path of the object file (DLL, shared library or bundle file) of dynamically loaded modules. For built-in handlers, this is an empty string.

- *orcid*
  The **ORCID** code of the author (see www.orcid.org).
- *path*
  The repository path for displaying hierarchical repository trees. This attribute is independent of any file system paths.
- *phone*
  A contact phone number of the author.
- *references*
  Cross references of the module. This is a nested list of class **UUID**s and reference type tags.
- *regid*
  For officially registered data type handlers, this is the assigned registration ID. Unregistered modules report zero.
- *slot*
  The slot in the handler table the module was loaded into.
- *sourcefile*
  The name of the source file for the module, if it is available.
- *sql*
  The **SQL** type name of the data type. In case there is no applicable **SQL** data type, this is an empty string.
- *sqlarray*
  The PostgresQL-style type name of **SQL** arrays of the data type. In case there is no such **SQL** type, this is an empty string.
- *version*
  Version information for the module. This is a string in a 1.2.3 (or shortened) style.
- *versionuuid*
  The **UUID** associated with this module version.
- *xml*
  The **XML** type name of the data type.

typex list

```
typex list ?pattern?
Typex.List(?pattern=?)
```

Return a list of all currently loaded data types, including those handled by built-in data type handlers. If desired, the list can be filtered by a string pattern.

typex load

```
typex load datatypename ?objectfile?
typex load all
Typex.Load(datatypename,?objectfile?)
Typex.Load("all")
```

Explicitly load a data type handler module. If the module is already loaded, the current version is unloaded first.

If no specific object file (a shared library on Unix/Linux, a DLL on Windows, a bundle file for MacOSX) is specified, the standard name of the module file is automatically generated from the data type name, and then the file searched in the directories in the data type handler module path. The module search path can be customized in the control variable **`::cactvs(typexpath)`**.

For **Tcl**, the return value of the command is the slot in the handler module table the module has been loaded into. This corresponds to the value of the *slot* attribute which can be queried via **`typex get`**. For **Python**, the return value is a module reference.

The magic name *all* instructs the program to traverse the data type extension search path and to load all data type extension modules found which are not yet loaded. In that case, the return value of the command is empty.

typex ref

```
Typex.Ref(datatypename)
```

Python-only method to get a reference of the module, which allows terser attribute retrieval commands and other operations.

typex subcommands

```
typex subcommands
dir(Typex)
```

Return a list of all subcommands of the **`typex`** command in the current interpreter.

typex unload

```
typex unload ?datatypename?...
tx.unload()
Typex.Unload(?txref/datatypename?,...)
```

Unload zero or more data type handler modules. It is an error to specify the name of a module which is not loaded. Built-in handler modules cannot be unloaded. Unloading a handler module when there is still property data of that type linked to chemical objects lead to memory leaks and/or confusing error messages when operating with these objects.

The return value is the number of unloaded modules.

The *ldap* Module

Various operations on **LDAP** directories are supported after loading the external **LDAP TCL** or **PYTHON** module. Only after loading the module, the **`ldap`** extension command becomes accessible.

Example for **TCL**:

```
package require Ldap
```

or, in **PYTHON**,

```
from ldap import *
```

The next step is usually the establishment of a connection (*bind*) to a known **LDAP** server. In many cases credentials are required for a successful bind with a DN containing a user name. After a completed *bind*, a handle or reference is returned. This handle can be used for further operations, such as information retrieval, until the connection is closed by an *unbind* command.

The general syntax of the **`ldap`** command follows the usual *command/subcommand/handle/ parameters* syntax.

Examples:

```
set lhandle [ldap bind $host $port $binddn $passwd]
ldap unbind $lhandle
```

The **LDAP** module currently only supports synchronous operations.

This is the list of subcommands:

ldap add

```
ldap add ldhandle dn attribute_list
l.add(dn=,attributes=)
```

Add one or more attributes to an existing DN. Other than that the default mode is **`LDAP_MOD_ADD`**, this command is identical to **`ldap replace`**.

The command returns the original handle or reference.

ldap addrdn

```
ldap addrdn ldhandle dn newrdn
l.addrdn(dn=,newdn=)
```

Add a new relative DN to an existing **LDAP** directory identified by the base DN.

The command returns the original handle or reference.

ldap bind

```
ldap bind host ?port? ?bindDN? ?password?
ldap open host ?port? ?bindDN? ?password?
Ldap(host=,?port=?,?bindDN=?,?password=?)
Ldap.Bind(host=,?port=?,?bindDN=?,?password=?)
```

Open a connection to an **LDAP** server and return the connection handle. Only the host parameter is required. If no port is specified, or an empty string, the default **LDAP** port (389) is used. If no distinguished name for binding is supplied, or an empty string, an anonymous bind is attempted. If an access control password is required, it may be supplied as last parameter.

The `open` command alternative is simply an alias intended to provide the standard command nomenclature for opening connections or files.

Example:

```
set lh [ldap bind $host 389 "cn=mr_x,cn=users,dc=chemcodes,dc=com" $themagicword]
```

ldap close

This is an alias for **`ldap unbind`**.

ldap delete

```
ldap delete ldhandle ?dn?...
l.delete(?dn?,...)
```

This command deletes one or more DNs from the current **LDAP** server.

ldap get

```
ldap get ldhandle attribute
l.get(attribute)
l.attribute
l[attribute]
```

Query status information about the **LDAP** connection. Currently, the following attributes can be queried:

- *derefmode*
  Get the method for dereferencing aliases. The value must be one of *never*, *searching finding (*i.e when locating a base object, but not when searching under it), or *always*.
- *handle*
  The **Tcl** handle of the object.
- *sizelimit*
  The maximum number of returned responses. The default is 0, meaning no limit on the number of responses.
- *timelimit*
  A timeout value in seconds for server queries. The default is 0, meaning no time limit.

Example:

```
set tl [ldap get $ldhandle timelimit]
```

ldap list

```
ldap list ?pattern?
Ldap.List(?pattern=?)
```

List all currently open **LDAP** connections. A list of the connection handles is returned Optionally, the handles, which are of the form *ldap%d*, may be filtered by a string pattern.

Example:

```
ldap list
```

ldap ref

```
Ldap.Ref(identifier)
```

PYTHON-only method to get a reference of the **LDAP** object from its handle.

ldap replace

```
ldap replace ldhandle dn attribute_list
l.replace(dn=,attributes=)
```

This command can be used to add, replace or delete one or more attributes of a DN. The attribute list is a standard **TCL** list or **PYTHON** tuple, where each attribute is a list element. The list elements must be of the form *attribute=value*, or simply *attribute* for deletions. Optionally, they may be prefixed by one of the characters "+" (add attribute), "=" (replace attribute, or create new if not existing), and "-" (delete attribute). The implicit default is "=". It is possible to add multiple instances of attributes, such as a set of e-mail addresses in the form

```
set alist [list "+mail=wdi@chemcodes.com" "+mail=wdi@xemistry.com"]
```

The **ldap add** command is a variant of this command - the only difference is that the default modification mode is **LDAP_MOD_ADD** instead of **LDAP_MOD_REPLACE**, corresponding to am implicit "+" prefix instead of "=".

ldap update is a command alias.

The command returns the original handle or reference.

Example:

```
ldap replace $ldhandle "cn=wdi,cn=users,dc=chemcodes,dc=com" \
    "mail=wdi@chemcodes.com"
```

ldap replacerdn

```
ldap replacerdn ldhandle dn newrdn
l.replacerdn(dn=,newdn=)
```

Replace a DN in an **LDAP** directory. The old DN is deleted, and replaced by the new DN, which may be specified relative to the old.

ldap updaterdn is a command alias.

The command returns the original handle or reference.

ldap search

```
ldap search ldhandle baseDN ?scope? ?getdn? ?filter? ?attributelist?
l.search(basedn=,?scope=?,?getdn=?,?filter=?,?attributes=?)
```

Query an **LDAP** server for information. Besides an active **LDAP** connection, the minimal argument set is only the base DB, the root from where the query should start on the connected server.

The optional scope argument may be *default* or an empty string (the default setting, in **PYTHON** you can also use **None**), *base* (search only the base object), *one* or *level* (search one level of sub-objects), or *subtree* (can be shortened to *sub*, search all sub-objects of the base object).

The *getdn* parameter is a boolean flag. If it is set, the first element of each response sublist is the DN of the entry. By default, only the attribute data is returned.

The filter parameter is a standard **LDAP** filter, which can be used to select subsets of directory entries. An empty string in this position (or `None` for **PYTHON**), or omitting the parameter, disables filtering.

Finally, the last parameter is the set of attributes which should be returned. If it is omitted, or an empty string (or `None` for **PYTHON**), all attributes of each matched entry are returned. If a requested attribute is not present in a matching record, it is silently omitted from the result list.

If no errors occurred, the result is a triply nested list. The outermost list contains one element for each entry. If a maximum number of responses was set to a positive value (*sizelimit* configuration parameter, see **`ldap set`** command), the maximum number of list elements is defined by this parameter. Each outer list element is itself a list. The middle lists contain one element for each returned attribute. Each of these is formatted as another sublist with *attribute* and *value* list elements. The attributes are returned in the order they were specified, provided that they are found in the returned set. If requested attributes are not present, they are silently omitted from the result list.

The command currently does support the retrieval of binary data.

Example:

```
ldap search $ldhandle "cn=users,dc=chemcodes,dc=com" one 0 {} [list mail phone]
```

returns a nested list of addresses, in the format

```
{{mail user1@addr1} {phone xxx-100}} {{mail user2@addr2} {phone xxx-105}}...
```

if both the *mail* and *phone* attributes are present.

The retrieval of the e-mail address of user *wdi* may be achieved either by using a filter, or a more specific DN:

```
ldap search $ldhandle "cn=users,dc=chemcodes,dc=com" one 0 cn=wdi "mail phone"
ldap search $ldhandle "cn=wdi,cn=users,dc=chemcodes,dc=com" one 0 {} "mail phone"
```

The filter argument[4] is a string representation of the filter to apply in the search. Simple filters can be specified as *attributetype=attributevalue*. More complex filters are specified using a prefix notation according to the following **BNF**:

```
<filter> ::= '(' <filtercomp> ')'
<filtercomp> ::= <and> | <or> | <not> | <simple>
<and> ::= '&' <filterlist>
<or> ::= '|' <filterlist>
<not> ::= '!' <filter>
<filterlist> ::= <filter> | <filter> <filterlist>
<simple> ::= <attributetype> <filtertype> <attributevalue>
<filtertype> ::= '=' | '~=' | '<=' | '>='
```

The '~=' construct is used to specify approximate matching. The representation for **`<attributetype>`** and **`<attributevalue>`** are as described in RFC 2254. In addition, **`<attributevalue>`** can be a single * to achieve an attribute existence test, or can contain text and *'s interspersed to achieve substring matching.

For example, the filter *"mail=*"* finds any entries that have a mail attribute. The filter *"mail=*@terminator.rs.itd.umich.edu"* will find any entries that have a mail attribute ending in the specified string. To put parentheses in a filter, escape them with a backslash '\' character. See RFC 2254 for a more complete description of allowable filters.

4. Description of filters copied from the OpenLDAP man page for ldap_search(3)

ldap set

```
ldap set ldhandle ?attribute value?...
ldap set ldhandle dict
l.set(?attribute,value?,...)
l.set(dict)
l.set(attribute=value,...)
l.attribute = value
l[attribute] = value
```

Set one or more attributes for an **LDAP** connection. Currently, the following attributes can be set:

- *derefmode*
  Set the method for dereferencing aliases. The value must be one of *never*, *searching finding (*i.e when locating a base object, but not for searching under it), or *always*.
- *sizelimit*
  The maximum number of response record to accept. If set to 0, any number of responses are accepted.
- *timelimit*
  The maximum time in seconds to wait for a server response. If set to 0, no timeout occurs.

The command returns the original handle or reference.

Example:

```
ldap set $ldhandle sizelimit 5 timelimit 500
```

ldap unbind

```
ldap unbind ldhandle ?ldhandle?...
ldap close ldhandle ?ldhandle?...
ldap unbind all
ldap close all
l.unbind()
l.close()
Ldap.Unbind(?lref/ldhandle?,...)
Ldap.Close(?lref/ldhandle?,...)
Ldap.Unbind("all")
Ldap.Close("all")
```

The first variant *unbinds* or *closes* (these are equivalent commands) a specific set of **LDAP** connections. All resources associated with the connection are freed, and the handles invalidated. However, they may later be reassigned to new connections.

The second alternative closes all currently opened **LDAP** connections.

The command returns the number of closed connections.

Example:

```
ldap close all
```

ldap verify

```
ldap verify ldhandle bindDN password
ldap verify host port bindDN password
```

```
l.verify(dn=,password=)
Ldap.Verify(host=,port=,dn=,password=)
```

Perform a basic (**`LDAP_AUTH_SIMPLE`**) user/access verification. The first variant uses an existing handle and attempts to re-bind it with a different distinguished bind name. The connection remains bound to the new address and DN.

The second variant temporarily creates a new **LDAP** connection and attempts to bind. The parameters have the same meaning as in the **`ldap bind`** command. The status is saved and then the connection is immediately closed. No persistent **LDAP** object is created.

This command returns boolean 1 for a successful bind, and 0 for failure.

Example:

```
ldap verify $host "" "dn=wdi,dn=users,dc=chemcodes,dc=com" $mypwd
```

The *gdbm* Module

The **gdbm** extension command is provided by the external **Gdbm** **Tcl** module. It must be loaded before the command become available.

Example for **Tcl**:

```
package require Gdbm
```

and for **Python**

```
from gdbm import *
```

Gdbm files need to be opened or created before any operations on them can commence. Afterwards, the access object is identified by a handle, which is returned by the opener commands. When they are no longer in use, they should be closed. **Gdbm** files have different internal structure on 32 vs. 64 bit systems, and are byte-order-dependent. They are therefore best suited for local, temporary files.

The general syntax of the **gdbm** command follows the usual *command/subcommand/handle/parameters* syntax. The **gdbm** command is to a large degree compatible to the newer and generally preferred **tc** command, which performs the same types of operations on **Tokyo Cabinet** files. The latter command can be significantly faster for large files, generated more compact data files. Additionally, these data files are 32/64-bit clean and byte-order-independent and thus much more portable between systems.

Note that the **Python** version of this module is not identical to the standard **Python** *gdbm* module.

Examples:

```
set gdhandle [gdbm open mygdbmfile.gdb]
set data [gdbm get $gdhandle $key]
gdbm close $gdhandle
```

This is the list of subcommands:

gdbm add

```
gdbm add gdhandle ?-nocase? key ?data?...
g.add(key=,data=,?nocase=?)
```

Append the listed data items as **Tcl** list elements to the entry identified by the key. If no such record exists, a new record with the initial set of list items is created if there are data arguments. If any of the data items are already present in the current value list, they are ignored, so duplicates are not added. The duplicate check is performed in case-insensitive fashion if the *-nocase* flag is used.

The **gdbm append** command performs a similar operation, but without a duplicate check.

The command returns the updated entry value list.

This command only works if an existing value is a properly formatted **Tcl** list. For **Python**, you can either pass a list or tuple, or a string as data argument which will be split into list items according to the **Tcl** syntax.

Example:

```
gdbm add $gdhandle "project_$projectid" [ens get $eh E_IDENT]
gdbm add $gdhandle "project_$projectid" [ens get $eh E_IDENT]
```

The second statement has no effect.

gdbm append

```
gdbm append gdhandle key ?data?...
g.append(key=,data=)
```

Append the data items as **Tcl** list elements to the entry identified by the key. If no such record exists, a new record with the initial set of list items is created if there are any data arguments. No duplicate check is performed for the added list elements - this is the difference to the **`gdbm add`** command.

The command returns the updated entry value list.

This command only works if an existing value is a properly formatted **Tcl** list. For **Python**, you can either pass a list or tuple as data argument, or a string which will be split into list items according to the **Tcl** syntax.

Example:

```
gdbm append $gdhandle "project_$projectid" [ens get $eh E_IDENT]
gdbm append $gdhandle "project_$projectid" [ens get $eh E_IDENT]
```

The second statement adds a duplicate list element to the entry.

gdbm close

```
gdbm close ?gdhandle?...
gdbm close all
g.close()
Gdbm.Close(?gref/ghandle?,...)
Gdbm.Close("all")
```

This command closes opened **Gdbm** files. After a file has been closed, its handle becomes invalid, but may the reissued for another opened **Gdbm** file again.

The first version of the command closes specific files. The second version closes all open **Gdbm** files in the current application. Both variants return the number of closed files as result.

Example:

```
gdbm close $gdhandle
```

gdbm count

```
gdbm count gdhandle ?pattern?
g.count(?pattern=?)
```

Count the number of keys in the file. If a pattern argument is given, only the file entries with a matching key are counted. The command returns the number of passing keys.

Unfortunately, **Gdbm** files do not maintain an internal record count, so this command has to loop through all keys, which can take a substantial amount of time for large files.

The command **`gdbm size`** is an alias for this command.

Example:

```
set size [gdbm count $gdhandle]
```

gdbm create

```
gdbm create filename ?mode? ?filemode? ?blocksize?
Gdbm.Create(filename=,?mode=?,?filemode=?,?blocksize=?)
Gdbm(filename=,?mode=?,?filemode=?,?blocksize=?)
```

This command is the same as the command **gdbm open**, except that the default file access mode to be used if no explicit mode is specified is *create* instead of *read*.

Example:

```
set gdhandle [gdbm create thefile.gdb]
```

gdbm delete

```
gdbm delete gdhandle key ?ispattern?
g.delete(key=,?ispattern=?)
```

If no *ispattern* boolean argument is given, or it is not a *true* value, the record matching the key exactly is deleted. If the entry did exist and could be deleted, boolean 1 is returned, 0 otherwise.

If the *ispattern* flag is set, the key argument is interpreted as a string match pattern. All records with keys matching the pattern are deleted. In this case, the return value is the number of deleted entries.

Example:

```
gdbm delete $gdhandle count* 1
```

deletes all records with keys starting with *count*.

gdbm dump

```
gdbm dump gdhandle ?file/pipe/std_channel/tcl_channel? ?keypattern? ?datapattern?
g.dump(?filename=?,?keypattern=?,?datapattern=?)
```

If no file or channel handle is specified, or an empty string is used (or **None** for **Python**), this command is used to obtain the complete contents of a **Gdbm** file as a dictionary. If non-empty filter patterns are set, only those key/value pairs where the key or data part matches the respective pattern are reported.

If a file name, a Unix pipe in standard notation, a standard channel or a valid **Tcl** channel handle argument is specified, and the target is writable, the filtered key/value pairs are written to that channel. Every key/value pair is formatted as a simple two-element **Tcl** list (even in the **Python** interface) and written to the file. In this mode, the return value is the number of lines written.

Example:

```
gdbm dump $gdhandle stdout
```

gdbm exists

```
gdbm exists gdhandle key
g.exists(key)
```

This command returns boolean 1 if the key is in the **Gdbm** file, 0 otherwise.

Example:

```
set isknown [gdbm exists $gdhandle [ens get $ehandle E_HASHY]]
```

gdbm first

```
gdbm first gdhandle ?pattern?
g.first(?pattern=?)
```

Returns the key of the first file entry. In case a pattern is specified, the first entry whose key matches the pattern is retrieved. If no matching or first key can be found, an error results.

Example:

```
gdbm first $gdhandle Z*
```

returns the first key which starts with a (case-sensitive) Z. Keys are returned in an unpredictable order.

This command is typically used in combination with subsequent **`gdbm next`** commands.

gdbm get

```
gdbm get gdhandle key ?silent? ?defaultvalue?
g.get(key=,?silent=?,?default=?)
g.key
g[key]
```

Retrieve the data value associated with the specified key. If the key does not exist, and a default value was specified, the default value is returned instead. Without a default value, the result is an empty string (**Tcl**) or None (**Python**) if the *silent* flag is set, and an error otherwise.

This command may return binary data with non-printable characters, so the language objects are byte arrays, not strings.

For **Python**, if the key is the same as the name of one of the class methods, only **`g.get()`** can be used to retrieve the data.

Example:

```
gdbm get $gdhandle [gdbm first $gdhandle]
```

gdbm incr

```
gdbm incr gdhandle key ?delta?
g.incr(key=,?delta=?)
```

Increment the data value stored under the specified key by the delta value. If no delta value is specified, it defaults to one. If the key does not exist, a new key/value pair with an initial data value of the delta is created. If the entry exists, but the data value is not a valid integer, an error is raised. The value is stored as a string.

The return value of this command is the incremented value.

Example:

```
gdbm insert $gdhandle "count_dracula" 5
puts [gdbm inc $gdhandle "count_dracula"]
```

This command sequence updates the file, and outputs 6.

gdbm index

```
gdbm index gdhandle index ?pattern?
g.index(index=,?pattern=?)
```

Return the *nth* key in the database if no pattern is specified, or the *nth* key which matches the pattern. The key index starts with zero.

Example:

```
gdbm index $gdhandle 10 cpdname*
```

returns the key for the eleventh file record which starts with the string *cpdname*. If no such key exists, an error is raised.

gdbm insert

```
gdbm insert gdhandle ?key value?...
gdbm insert gdhandle dict
g.insert(?key,value?,...)
g.insert(dict)
```

Insert one or more new key/value pairs into the file. The values may be binary data. If a key already exists, the return value for that key/value pair is boolean 0 and the old data is not overwritten. If a key is not in the database, boolean 1 is returned for that item pair and the data is stored in the file.

If only a single argument is used after the handle, it is expected to be a properly formed dictionary. In that case, all dictionary elements are processed as if they were spelled out as individual key/value pairs.

The return value is a list of boolean flags indicating success or failure for the operation on each key/value pair. No error is raised if a write operation fails.

Example:

```
gdbm insert $gdhandle [ens get $ehandle E_HASH] [ens get $ehandle E_GIF]
gdbm insert $gdhandle [array get ::params]
```

gdbm keys

```
gdbm keys gdhandle ?pattern?...
g.keys(?pattern=?)
```

This command returns a list of all keys in the file. Keys may optionally be filtered by one or more string match patterns. The **Python** version only supports a single pattern.

The related **`gdbm match`** command can be used to obtain a list of keys which are filtered by the value component and not the key name.

Example:

```
set keylist [gdbm keys $gdhandle project${id}*]
```

gdbm linkvar

```
gdbm linkvar gdhandle ?-loadall? ?-preserve? varname
```

This command establishes a link between a **Gdbm** file and a **Tcl** array variable. If the variable is already in existence, it is deleted prior to recreation as a linked array variable. This can be prevented by using the *-preserve* option. This command is not supported with **Python**.

After the link has been formed, any read access by a **Tcl** script to an element of the array variable retrieves the data value from the file which corresponds to the array element name as key, if the variable element is not yet set. If the variable element already exists, it takes precedence over the file contents, but this can only happen if the *-preserve* option was used when the variable was linked. In case file retrieval is initiated, and the key does not exist in the file, an error is generated.

Assigning a value to an array element replaces or creates the corresponding file entry. If an array element is deleted (for example by the **Tcl** **unset** command), the corresponding key and value in the file are also deleted. Replacement and deletion operations require a file opened for write access.

If the *-loadall* option is used, all array elements are immediately loaded by looping over all file keys when the command is run. By default, data is retrieved from the file only when an array element is explicitly accessed. If this option is used, the **array names** command immediately shows all file keys, not just the ones currently loaded.

Since **Gdbm** data, once retrieved from the file by accessing the linked array element, is never deleted from memory, the use of this utility is not recommended for large files.

It is possible to link multiple variables to a single **Gdbm** file.

Gdbm files must not be closed when variable links to the closed file are active. Currently, there is no mechanism to detect that the target of a variable link as gone away, so this cannot be handled automatically. Variable links can be removed by the **gdbm unlinkvar** command.

Example:

```
gdbm linkvar $gdhandle g_array
puts "Key: $thekey Value: $g_array($thekey)"
set g_array($newkey) $newvalue
unset g_array($thekey)
```

gdbm list

```
gdbm list gdhandle ?pattern?
Gdbm.List(?pattern=?)
```

Get a list of currently active **Gdbm** file handles. If no files have been opened, an empty list is returned. Optionally, a string filter pattern can be specified.

Example:

```
gdbm list
```

gdbm loop

```
gdbm loop gdhandle keyvar datavar ?pattern? body
g.loop(function=,?keyvariable=?,?datavariable=?,?pattern=?)
```

Run a loop over all file entries. If a string filter pattern is specified, only those records with a matching key are visited. After retrieving the key and data values, the key and data variables are initialized to the current key and data values, and then the commands in the body section are executed.

With **Tcl**, in the body, the standard *break* or *return* statements may be used to force an early exit from the loop, and the *continue* statement also works as expected. In case an uncaught error occurs in the body, the loop terminates with an error message.

The **Python** interface intentionally has a different argument sequence. The function may either be a function reference, or a multi-line string containing the loop body code similar to the **Tcl** style. If a function reference is used, the function is called with a key/value tuple as only argument. Loop variables are not required in this mode (though they are still supported). Normal *break* and *continue* loop control statements cannot be used in the **Python** functions. Instead, the custom *BreakLoop* and *ContinueLoop* exceptions can be thrown for the same effect.

If the loop was not terminated due to an error, the result value is the number of visited keys.

Example:

```
gdbm loop $gdhandle key data {
   puts "Key: $key with value $data"
}
```

gdbm match

```
gdbm match gdhandle ?pattern?
g.match(?pattern=?)
```

This command returns a list of all keys where the data value matches the filter pattern. This is similar to the **`gdbm keys`** command, with the difference that in that command the key names are used for filtering, not the data values.

Example:

```
set keylist [gdbm match $gdhandle *nitro*]
```

gdbm new

```
gdbm new filename ?mode? ?filemode? ?blocksize?
Gdbm.New(filename=,?mode=?,?filemode=?,?blocksize=?)
```

This command is the same as the command **`gdbm open`**, except that the default file access mode to be used if no explicit mode is specified is *new* instead of *read*.

Example:

```
set gdhandle [gdbm new thefile.gdb]
```

gdbm next

```
gdbm next gdhandle key ?pattern?
g.next(key=,?pattern=?)
```

Get the next key after the specified key which matches the pattern. If no pattern is specified, no key filtering is performed. If no key can be found, an error is raised, otherwise the key is returned as function result.

A starting point for a key traversal is usually obtained via a **`gdbm first`** or **`gdbm index`** command.

Example:

```
gdbm next $gdhandle name20 name*
```

produces the next key after the current key *name20* matching the pattern *name**. This key is not necessarily *name21*, or any other predictable value. Rather, the key sequence in the database is determined by its internal hash, and the ordering is pseudo-random. The only guarantee is that with a **`gdbm first/gdbm next`** loop all keys are ultimately visited. A deletion of the current key is allowed and guaranteed to not disturb the traversal sequence.

gdbm open

```
gdbm open filename ?mode? ?filemode? ?blocksize?
```

```
Gdbm(filename=,?mode=?,?filemode=?,?blocksize=?)
Gdbm.Open(filename=,?mode=?,?filemode=?,?blocksize=?)
```

Open an existing **Gdbm** file, or create a new one. The command returns a new object handle or reference. The default mode is *read*, opening the file for reading. In this case, the file must already exist, and have been written on a computer with the same byte ordering and integer size. The *mode* parameter may consist of a bitset of the following options:

- *read*
  Open for reading only, the file must exist.
- *write*
  Open for reading and writing, the file must exist
- *create*
  Open for reading and writing, the file is created if it does not yet exist. The contents of an existing file are not modified.
- *new*
  Open for reading and writing, and the file must not yet exist. If it does, an error results.
- *fast*
  Adds the *fast* attribute - file updates are not committed immediately to disk, at the risk of losing data or file corruption if the program exits or crashes before the file is properly closed, or a **gdbm sync** command is issued.

It is sufficient to specify the first letter of the mode options. Combining any of the first four file access modes does not make sense, though. In standard applications, the mode set is either a single word describing the access mode, or a combination of the *fast* attribute and the access mode.

The optional *filemode* parameter determines the Unix-style octal file access bits which have an effect only in case a new disk file is created. Finally, the rarely used *blocksize* parameter determines the page block size of the **Gdbm** file and is again only used in case a new file is created. By default the value is 0, meaning that a reasonable default value depending on the type of file system the file resides on is chosen.

This command returns a **Gdbm** file handle in the form *gdbm%d* which can be used in subsequent **gdbm** commands, until the file is closed and the handle becomes invalid.

gdbm ref

```
Gdbm.Ref(identifier)
```

Python-only method to get a reference of the **Gdbm** object from its handle.

gdbm reorganize

```
gdbm reorganize gdhandle
g.reorganize()
```

Compact the data file. This can be useful after many deletions. The underlying **Gdbm** library copies the entries to another file, which ultimately replaces the original file. Thus, in worst case, sufficient disk space for two parallel files with the original file size must be available. This command requires write access to the **Gdbm** file.

gdbm reorg is an alias.

For large files, this command can take a long time to complete.

Example:

```
gdbm reorg $gdhandle
```

gdbm replace

```
gdbm replace gdhandle ?key value?...
gdbm replace gdhandle dict
g.replace(?key,value?,...)
g.replace(dict)
g.key = value
g[key] = value
```

Store one or more specified values which may be binary data, under the given keys. If a key is already in use, the old value is overwritten. If only a single argument is used after the handle, it is expected to be a properly formed dictionary. In that case, all dictionary elements are processed as if they were written out as individual key/value pairs.

The return value is a list of boolean flags indicating success or failure for the operation on each key/value pair. No error is raised if a write failed.

`gdbm set` is an alias for this command.

Examples:

```
gdbm replace $gdhandle $thekey $newdata $key2 $moredata
gdbm replace $gdhandle [array get ::params]
```

gdbm restore

```
gdbm restore gdhandle filename ?callback?
g.restore(filename=,?function=?)
```

This command reads a text file which usually was produced by the **`gdbm dump`** command and adds its contents to the current **GDBM** file. Every line of the file is expected to contain a properly formatted two-element **TCL** list. The list is split into the key and data parts and the contents are written to the **GDBM** file, replacing old entries in case of existing keys.

The input file may be gzip-compressed or plain **ASCII** text.

If a callback function is specified, it is called with the **GDBM** handle (or reference), the input file name, and the current restored item count after each successful line input.

The command **`gdbm readfile`** is an alias for this command.

The return value is the number of file lines read.

Example:

```
gdbm restore $gdhandle "we_will_never_need_this_backup.txt"
```

gdbm set

This is an alias for **`gdbm replace`**.

gdbm sync

```
gdbm sync gdhandle
g.sync()
```

Synchronize the in-memory and disk status of the file. All pending changes are committed to disk. This command has an effect only if the file was opened in fast mode (see **`gdbm open`** command).

`gdbm synchronize` is an alias.

Example:

```
gdbm sync $gdhandle
```

gdbm unlinkvar

```
gdbm unlinkvar gdhandle ?-preserve? varname
```

Unlink a **Tcl** array variable from a **Gdbm** file. If the *-preserve* option is not used, the variable is also deleted from the **Tcl** interpreter. The association between array variable and **Gdbm** file is also automatically broken when the variable is deleted by other means, such as a **Tcl** **`unset`** command.

The **Gdbm** file contents are not modified by unlinking by means of executing this command, or by any other methods of variable deletion.

Variable links are created with the **`gdbm linkvar`** command.

This command is not supported in **Python**.

Example:

```
gdbm unlinkvar $gdhandle g_array
```

The *tc* Module

The **tc** (**TOKYO CABINET**) extension command is provided by the external **Tc TCL** module. It must be loaded before the command becomes available.

Example (**TCL**):

```
package require Tc
```

or in **PYTHON**

```
from tc import *
```

TOKYO CABINET files need to be opened or created before any operations on them can commence. Afterwards, they are identified by a handle, which is returned by the opener commands. When a file are no longer in use, it should be closed.

The general syntax of the **tc** command follows the usual *command/subcommand/handle/parameters* syntax. This command is designed to be compatible to the older **gdbm** command, which performs the same type of operations on **GDBM** files. **GDBM** files generally require significantly more disk space, and are slower for larger data sets. Another big advantage or **Tc** versus **GDBM** is that the **Tc** files are 32/64-bit-clean and byte-order independent and thus are more easily exchanged between systems. On the other hand, the underlying **TOKYO CABINET** library is currently not supported on Windows, and therefore this command is also unavailable on that platform

At this time, only **Tc** hash data bases are supported, and this **TOKYO CABINET** database type is implied in all commands.

Examples:

```
set tchandle [tc open myfile.tch]
set data [tc get $tchandle $key]
tc close $tchandle
```

This is the list of of subcommands:

tc add

```
tc add tchandle ?-nocase? key ?data?...
t.add(key=,data=,?nocase=?)
```

Append the listed data items as **TCL** list elements to the entry identified by the key. If no such record exists, a new record with the initial set of list items is created if any data arguments are passed. If any of the data items are already present in the current value list, they are ignored, so duplicates are not added. The duplicate check is performed in case-insensitive fashion if the *-nocase* flag is used.

The **tc append** command performs a similar operation, but without a duplicate check.

The command returns the updated entry value list.

This command only works if an existing value is a properly formatted **TCL** list. For **PYTHON**, you can either pass a list or tuple as data argument, or a string which will be split into list items according to the **TCL** syntax.

Example:

```
tc add $tchandle "project_$projectid" [ens get $eh E_IDENT]
tc add $tchandle "project_$projectid" [ens get $eh E_IDENT]
```

The second statement has no effect.

tc append

```
tc append tchandle key data ?data?...
t.append(key=,data=)
```

Append the data items as TCL list elements to the entry identified by the key. If no such record exists, a new record with the initial set of list items is created if any data arguments are passed. No duplicate check is performed for the added list elements - this is the difference to the **`tc add`** command.

The command returns the updated entry value list.

This command only works if an existing value is a properly formatted TCL list. For PYTHON, you can either pass a list or tuple as data argument, or a string which will be split into list items according to the TCL syntax.

Example:

```
tc append $tchandle "project_$projectid" [ens get $eh E_IDENT]
tc append $tchandle "project_$projectid" [ens get $eh E_IDENT]
```

The second statement adds a duplicate list element to the entry.

tc close

```
tc close ?tchandle?...
tc close all
t.close()
Tc.Close(?tcref/tchandle?,...)
Tc.Close("all")
```

This command closes opened TOKYO CABINET files. After a file has been closed, its handle becomes invalid, but may the reissued for another opened TOKYO CABINET file again.

The first version of the command closes specific files. The second version closes all open TOKYO CABINET files. Both variants return the number of closed files as result.

Example:

```
tc close $tchandle
```

tc count

```
tc count tchandle ?pattern?
t.count(?pattern=?)
```

Count the number of keys in the file. If a pattern argument is given, only the file entries with a matching key are counted. The command returns the number of passing keys.

Unfortunately, TOKYO CABINET files do not maintain an internal record count, so this command has to loop through all keys, which can take a substantial amount of time for large files.

The command **`tc size`** is an alias for this command.

Example:

```
set size [tc count $tchandle]
```

tc create

```
tc create filename ?mode? ?filemode?
Tc.Create(filename=,?mode=?,?filemode=?)
```

This command is the same as the command **`tc open`**, except that the default file access mode to be used if no explicit mode list is specified is *create* instead of *read*.

Example:

```
set tchandle [tc create thefile.tch]
```

tc delete

```
tc delete tchandle key ?ispattern?
t.delete(key=,?ispattern=?)
```

If no *ispattern* boolean argument is given, or it is not a *true* value, the record matching the key exactly is deleted. If the entry did exist and could be deleted, 1 is returned, 0 otherwise.

If the *ispattern* flag is set, the key argument is interpreted as a string pattern. All records with keys matching the pattern are deleted. In this case, the return value is the number of deleted records.

Example:

```
tc delete $tchandle count* 1
```

deletes all records with keys starting with *count*.

tc dump

```
tc dump tchandle ?file/pipe/std_channel/tcl_channel? ?keypattern? ?datapattern?
t.dump(filename=,?keypattern=?,?datapattern=?)
```

If no file handle is passed, or an empty string (or **`None`** for **Python**) is used as file handle, this command is used to obtain the complete contents of a **Tokyo Cabinet** file as a dictionary. If non-empty filter patterns are specified, only those key/value pairs where the key or data part matches the respective pattern are reported.

If a file name, a Unix pipe in standard notation, a standard channel or a valid **Tcl** channel handle argument is specified, and the target is writable, the filtered key/value pairs are written to that channel. Every key/value pair is formatted as a simple two-element **Tcl** list (even in the **Python** interface) and written to the file. In this mode, the return value is the number of lines written.

Example:

```
tc dump $tchandle stdout
```

tc exists

```
tc exists tchandle key
t.exists(key)
```

This command returns boolean 1 if the key is in the **Tokyo Cabinet** file, 0 otherwise.

Example:

```
set isknown [tc exists $tchandle [ens get $ehandle E_HASHY]]
```

tc first

```
tc first tchandle ?pattern?
```

```
t.first(?pattern=?)
```

Returns the key of the first file entry. In case a pattern is specified, the first entry whose key matches the pattern is retrieved. If no key can be found, an error results.

Example:

```
tc first $tcandle Z*
```

returns the first key which starts with a (case-sensitive) Z. Keys are returned in an unpredictable order.

This command is typically used in combination with subsequent **`tc next`** commands.

tc get

```
tc get tchandle key ?silent? ?defaultvalue?
t.get(key=,?silent=?,?default=?)
t.key
t[key]
```

Retrieve the data value associated with the specified key. If the key does not exist, and a default value was specified, the default value is returned instead. Without a default value, the result is an empty string (**TCL**) or None (**PYTHON**) if the *silent* flag is set, and an error otherwise.

This command may return binary data with non-printable characters, so the language objects are byte arrays, not strings.

For **PYTHON**, if the key is the same as the name of one of the class methods, only **`t.get()`** can be used to retrieve the data.

Example:

```
tc get $tchandle [tc first $tchandle]
```

tc incr

```
tc incr tchandle key ?delta?
t.incr(key=,?delta=?)
```

Increment the data value stored under the specified key by the delta value. If no delta value is specified, it defaults to one. If the key does not exist, a new key/value pair with a data value of the delta is created. If the entry exists, but the data value is not a valid integer, an error is raised. The value is stored as a string.

The return value of this command is the incremented value.

Example:

```
tc insert $tchandle "count_dracula" 5
puts [tc incr $tchandle "count_dracula"]
```

This command sequence updates the file, and outputs 6.

tc index

```
tc index tchandle index ?pattern?
t.index(key=,?pattern=?)
```

Return the *nth* key in the database if no pattern is specified, or the *nth* key which matches the pattern. The key index starts with zero.

Example:

```
tc index $tchandle 10 cpdname*
```

returns the key for the eleventh file record which starts with the string *cpdname*. If no such key exists, an error is raised.

tc insert

```
tc insert tchandle ?key value?...
tc insert tchandle dict
t.insert(?key,value?,...)
t.insert(dict)
```

Insert one or more new key/value pairs into the file. The values may be binary data. If the key already exists, the return value for that key/value pair is 0 and the old data is not overwritten. If the key is not in the database, 1 is returned for the item pair and the value is stored in the file.

If only a single argument is used after the handle, it is expected to be a properly formed **Tcl** dictionary. In that case, all dictionary elements are processed as if they were spelled out as individual key/value pairs.

The return value is a list of boolean flags indicating success or failure for the operation on each key/value pair. No error is raised if a write operation fails.

Example:

```
tc insert $tchandle [ens get $ehandle E_HASH] [ens get $ehandle E_GIF]
tc insert $tchandle [array get ::params]
```

tc keys

```
tc keys tchandle ?pattern?...
t.keys(?pattern=?)
```

This command returns a list of all keys in the file. Keys may optionally be filtered by one or more string match patterns.

The related **`tc match`** command can be used to obtain a list of keys which are filtered by the value component and not the key name.

Example:

```
set keylist [tc keys $tchandle project${id}*]
```

tc linkvar

```
tc linkvar tchandle ?-loadall? ?-preserve? varname
```

This command establishes a link between a **Tokyo Cabinet** file and a **Tcl** array variable. If the variable is already in existence, it is deleted prior to recreation as a linked array variable. This can be prevented by using the *-preserve* option.

The command is not supported in **Python**.

After the link has been formed, any read access by a **Tcl** script on an element of the array variable retrieves the data value from the **Tokyo Cabinet** file which corresponds to the array element name as key, if the variable element is not yet set. If the variable element already exists, it takes precedence over the file contents, but this can only happen if the *-preserve* option was used when the variable was linked. In case file retrieval is performed, and the key does not exist in the file, an error is generated.

Assigning a value to an array element replaces or creates the corresponding **Tokyo Cabinet** file entry. If an array element is deleted (for example, **`by a Tcl unset`** command), the corresponding key and value in the file are also deleted. Replacement and deletion operations require a **Tokyo Cabinet** file opened for write access.

If the *-loadall* option is used, all array elements are immediately loaded by looping over all **Tokyo Cabinet** file keys when the command is executed. By default, data is retrieved from the file only when an array element is explicitly accessed. If this option is used, the **`array names`** command immediately shows all file keys, not just the ones currently loaded.

Since **Tokyo Cabinet** data, once retrieved from the file by accessing the linked array element, is never deleted from memory, the use of this mechanism is not recommended for large files.

It is possible to link multiple variables to a single **Tokyo Cabinet** file.

Tokyo Cabinet files must not be closed when variable links to the file are active. Currently, there is no mechanism to detect that the target of a variable link as gone away, so this cannot be handled automatically. Variable links can be removed by the **`tc unlinkvar`** command.

Example:

```
tc linkvar $tchandle g_array
puts "Key: $thekey Value: $g_array($thekey)"
set g_array($newkey) $newvalue
unset g_array($thekey)
```

tc list

```
tc list tchandle ?pattern?
Tc.List(?pattern=?)
```

Get a list of currently active **Tokyo Cabinet** file handles or references. If no files have been opened, an empty list is returned. Optionally, a string filter pattern can be specified.

Example:

```
tc list
```

tc loop

```
tc loop tchandle keyvar datavar ?pattern? body
t.loop(function=,?keyvariable=?,?datavariable=?,?pattern=?)
```

Run a loop over all file entries. If a filter pattern is specified, only those records with a matching key are visited. After retrieving the key and data values, the key and data variables are initialized to the current key and data values, and then the commands in the body section are executed.

With **Tcl**, in the body, the standard *break* or *return* statements may be used to force an early exit from the loop, and the *continue* statement also works as expected. In case an uncaught error occurs in the body, the loop terminates with an error message.

The **Python** interface intentionally has a different argument sequence. The function may either be a function reference, or a multi-line string containing the loop body code similar to the **Tcl** style. If a function reference is used, the function is called with a key/value tuple as only argument. Loop variables are not required in this mode (though they are still supported). Normal *break* and *continue* loop control statements cannot be used in the **Python** functions. Instead, the custom *BreakLoop* and *ContinueLoop* exceptions can be thrown for the same effect.

If the loop was not terminated due to an error, the result value is the number of visited keys.

Example:

```
tc loop $tchandle key data {
   puts "Key: $key with value $data"
}
```

tc match

```
tc match tchandle pattern
t.match(pattern=)
```

This command returns a list of all keys where the data value matches the filter pattern. This is similar to the **`tc keys`** command, with the difference that in that command the key names are used for filtering, not the data values.

Example:

```
set keylist [tc match $tchandle *nitro*]
```

tc new

```
tc new filename ?mode? ?filemode?
Tc.New(filename=,?mode=?,?filemode=?)
```

This command is the same as the command **`tc open`**, except that the default file access mode to be used if no explicit mode list is specified is *new* instead of *read*.

Example:

```
set tchandle [tc new thefile.tch]
```

tc next

```
tc next tchandle key ?pattern?
t.next(?pattern=?)
```

Get the next key after the specified key which matches the pattern. If no pattern is specified, no key filtering is performed. If no key can be found, an error is raised, otherwise the key is returned as command result.

A starting point for a key traversal is usually obtained via a **`tc first`** or **`tc index`** command.

Example:

```
tc next $tchandle name20 name*
```

produces the next key after the current key *name20* matching the pattern *name**. This key is not necessarily *name21*, or any other predictable value. Rather, the key sequence in the database is determined by its internal hash, and the ordering is pseudo-random. The only guarantee is that with a **`tc first/tc next`** loop all keys are ultimately visited. A deletion of the current key is allowed and guaranteed to not disturb the traversal sequence.

tc open

```
tc open filename ?mode? ?filemode?
Tc(filename=,?mode=?,?filemode=?)
Tc.Open(ilename=,?mode=?,?filemode=?)
```

Open an existing **Tokyo Cabinet** file, or create a new one. The command returns a new object handle or reference. The default mode is *read*, opening the file for reading. In this case, the file must already exist. The *mode* parameter may consist of a collection of the following keywords:

- *read*
  Open for reading only, the file must exist.
- *write*
  Open for reading and writing, the file must exist
- *create*
  Open for reading and writing, the file is created if it does not yet exist. The contents of an existing file are not changed.
- *new*
  Open new file for reading and writing. An error results if the file already exists.
- *fast*
  Adds the *fast* attribute - file updates are not committed immediately to disk, at the risk of losing data or file corruption if the program exits or crashes before the file is properly closed, or a **`tc sync`** command is issued.
- *compressed*
  Stored data is *zlib*-compressed.

In standard applications, the mode list is either a single word describing the access mode, or a list of the *fast* or *compressed* attributes and the access mode.

The optional *filemode* parameter determines the Unix-style octal file access bits which have an effect only in case the file is created.

This command returns a **Tokyo Cabinet** file handle in the form *tc%d* which can be used in subsequent **`tc`** commands, until the file is closed and the handle becomes invalid.

tc ref

```
Tc.Ref(identifier)
```

Python-only method to get a reference of the **Tokyo Cabinet** object from its handle.

tc reorganize

```
tc reorganize tchandle
t.reorganize()
```

Compact the data file. This can be useful after many deletions. This command requires write access to the file.

`tc reorg` is an alias.

For large files, this command can take a long time to complete.

Example:

```
tc reorg $tchandle
```

tc replace

```
tc replace tchandle ?key value?...
tc replace tchandle dict
t.replace(?key,value?,...)
t.replace(dict)
t.key = value
t[key] = value
```

Store one or more specified values which may be binary data, under the given keys. If a key is already in use, the old value is overwritten. If only a single argument is used after the handle, it is expected to be a properly formed **Tcl** dictionary. In that case, all dictionary elements are processed as if they were spelled out as individual key/value pairs.

The return value is a list of boolean flags indicating success or failure for the operation on each key/value pair. No error is raised if a write failed.

`tc set` is an alias for this command.

Examples:

```
tc replace $tchandle $thekey $newdata $key2 $moredata
tc replace $tchandle [array get ::params]
```

tc restore

```
tc restore tchandle filename ?callback?
t.restore(filename=,?function=?)
```

This command reads a text file which usually was produced by the **`tc dump`** command and adds its contents to the current **Tokyo Cabinet** file. Every line of the file is expected to contain a properly formatted two-element **Tcl** list. The list is split into the key and data parts and the contents are written to the Tokyo Cabinet file, replacing old entries in case of existing keys.

The input file may be gzip-compressed or plain ASCII text.

If a callback function is specified, it is called with the object handle (or reference), the input file name, and the current restored item count after each successful line input.

The command **`tc readfile`** is an alias for this command.

The return value is the number of file lines read.

Example:

```
tc restore $tchandle "totally_superfluous_backup.txt"
```

tc set

This is an alias for **`tc replace`**.

tc sync

```
tc sync tchandle
t.sync()
```

Synchronize the memory and disk status of the file. All pending changes are committed to disk. This command has an effect only if the file was opened in fast mode (see **`tc open`** command).

`tc synchronize` is an alias.

Example:

```
tc sync $tchandle
```

tc unlinkvar

```
tc unlinkvar tchandle ?-preserve? varname
```

Unlink a **Tcl** array variable from a **Tokyo Cabinet** file. If the *-preserve* option is not used, it is also deleted from the **Tcl** interpreter. The association between array variable and **Tokyo Cabinet** file is also automatically broken when the variable is deleted by other means, such as a **Tcl** **`unset`** command.

This command is not supported in the **Python** interface.

The **Tokyo Cabinet** file contents are not modified by unlinking by means of executing this command or any other method of variable deletion.

Variable links are created with the **`tc linkvar`** command.

Example:

```
tc unlink $tchandle g_array
```

The memcache Command Extension

The **`memcache`** command extension provides an interface to the *memcached* memory data caching daemon. This is a useful functionality especially for the implementation of stateful **CGI** and **FCGI** applications. This module is now the preferred replacement for the older *netcache* daemon interface. Current toolkit packages contain both the interface module, and a compiled version of the daemon proper.

The command is either auto-loaded, or can be explicitly loaded by a statement like

```
cmdx load memcache
```

or (on **PYTHON**)

```
from memcache import *
```

The command is thread-safe (with the usual caveats for **PYTHON**).

These are the supported subcommands:

memcache append

```
memcache append mchandle data
memcache append mchandle key data ?ttl_secs? ?flags?
m.append(?key=?,?data=?,?ttl=?,?flags=?)
```

This command is a variant of the **`memcache put`** command which appends its data instead of replacing it, if a record already exists for the key. If no such record exists, the command is equivalent to **`memcache put`**.

For the explanation of the arguments, please refer to the paragraph on **`memcache put`** below.

The command returns the key of the tuple, which may have been assigned automatically.

Example:

```
memcache append $mch $key $moredata
```

memcache create

```
memcache create host:port/socketfile ?host:port/socketfile?...
Memcache(host:port/socketfile,?host:port/socketfile?,...)
Memcache.Create(host:port/socketfile,?host:port/socketfile?,...)
```

Create a new *memcached* interface object. The command returns a handle of the new object which is used by other memcache commands to identify the object. Multiple interface objects may be used in parallel in an application.

The arguments identify the servers to contact. In case multiple servers are listed, tuples are stored on or retrieved from one of the servers selected in constant but pseudo-random fashion from the key value, resulting in load-balancing.

Servers are identified either by a host name, optionally with an addition custom port number, or the name of a named socket in file system space. An argument is interpreted as a socket name if it contains path separator characters (“/” or “\”), starts with a dot, or starts/ends with a vertical bar “|” or equality sign “=”. If the bar or equality sign are present, it is automatically stripped from the actually used socket name. An attempt is then made to check whether the named socket already exists and can be accessed. If this is not the case, an attempt is made to set it up, and an error results if this operation does not succeed.

Alternatively, servers can be named in the classical *hostname:port* fashion, where the port part is optional. If no port is specified, the default *memcached* port 11211 is used. The host name of a server specified here must be resolvable, but at the moment the interface object is created it is not required that a daemon is already listening at the port.

Examples:

```
set mch [memcache create cachehost1 cachehost2]
set mch [memcache create ./mysocket]
```

memcache delete

```
memcache delete all
memcache delete ?mchandle?...
m.delete()
Memcache.Delete(?mchandle/mcref?,...)
Memcache.Delete("all")
```

The first command version deletes all active memcache interface objects, after properly closing their connections. The second variant closes and deletes specific handles. The command returns the number of deleted interface objects.

The functionality of the first command variant is automatically executed if the **`memcache`** command extension is unloaded.

Deleting an interface handle does not immediately delete the tuples created via this interface on the daemon. The data remains stored until its time-to-live expires, and may be fetched by other interface object instances, even in different processes on different hosts, if they connect to the same daemon and supply the proper keys. To delete tuples, use the **`memcache remove`** command.

Deleting an interface handle furthermore does not delete any named sockets associated with the handle, regardless whether these were already present when the object was created, or whether they were automatically set up as a side effect of the object creation.

Example:

```
memcache delete $mch
```

memcache get

```
memcache get mchandle key ?listindex?
m.get(key=,?index=?)
m.key
m[key]
```

Retrieve the data associated with the specified key. The data, which can be binary in nature, is returned as the command result. In case there is no tuple associated with the key, including due to expiration of the lifetime of data associated with the key on the remote daemon, an error results.

If the optional list index is specified, the data is expected to be a properly formed **TCL** list. If interpretation as list succeeds, only the selected list element is returned. If the list index is larger than the number of list items in the data, an empty string is the result, but no error is raised.

For **PYTHON**, if the key is the same as the name of one of the class methods, only **`m.get()`** can be used to retrieve the data.

Example:

```
set data [memcache get $mch $key]
```

memcache list

```
memcache list ?pattern?
Memcache.List(?pattern=?)
```

Get a list of currently used *memcache* object handles or references. If no *memcache* objects have been created, an empty list is returned. Optionally, a string filter pattern can be specified.

Example:

```
memcache list
```

memcache put

```
memcache put mchandle data
memcache put mchandle key data ?ttl_secs? ?flags?
m.put(?key=?,data=,?ttl=?,?flags=?)
m.key = data
m[key] = data
```

Store data on a *memcache* daemon. The data is stored as an opaque byte array and can be of any type. It is not necessary to base64-encode the data or protect it in any other fashion.

The first command variant automatically assigns an unique key and stores the data with it.

The second variant is more flexible. If the key parameter is an empty string, a new unique key is automatically generated. However, with an explicit key, data can be replaced if the key is already in use. If an explicitly specified key does not exist, a new tuple is silently created. Keys are also byte arrays and could potentially be binary data, but commonly some readable string encoding is used. Optionally, the time-to-live of the data on the daemon can be specified. The default lifetime is 8 hours. The **`TTL`** value is relative to the current time.

Tuples which are in the store longer than their lifetime are automatically retired by the daemon. Attempts to access them via their key after this point in time results in an error. The final optional argument is an integer, which can be stored as an additional tuple component.

The command returns the key of the tuple, which may have been assigned automatically.

Example:

```
set key [memcache put $nh $data1]
memcache put $nh $key $data2
```

memcache ref

```
Memcache.Ref(identifier)
```

PYTHON-only method to get a reference of the **MEMCACHE** object from its handle.

memcache remove

```
memcache remove mchandle key ?ttl_secs?
m.remove(key=,?ttl=?)
```

Delete a tuple on the remote daemon. If the key is not valid, an error results. By default the deletion becomes effective immediate. Alternatively a time-to-live value can be specified, which schedules tuple deletion for a time in the future. The **`TTL`** value is relative to the current time.

This command does not delete the interface object. For this purpose, use the **`memcache delete`** command.

Example:

```
catch {memcache remove $mch $key}
```

This statement removes the tuple, and ignores any errors in case the key is not stored or has already timed out

memcache start

```
memcached start mchandle
m.start()
```

Attempt to start all required *memcached* daemons associated with an interface object. If all connections are already being served, this command does nothing.

Otherwise, the command will try to start *memcached* executable(s) with properly configured options for a rendezvous on the named socket or socket port as specified during the interface object creation. This autostart mechanism obviously only works of the daemon is running on the same host as the application, the *memcached* executable is found in the search path, and the effective user of the application has sufficient permissions. Other start mechanisms must be used to assure that the required daemons are listening when using remote ports.

If there were unserved connections, and they could be activated by starting the daemon locally, the command result is 1. If any required start attempt failed, the result is 0. If the command did not need to perform any action, the result is -1.

Any newly started daemon has an empty tuple storage and no memory of any data stored by daemons previously serving the same connection.

memcache status

```
memcache status mchandle
m.status()
```

Get the status of all daemons connected to the handle. If there is only a single daemon associated with the interface object, the result is a status dictionary with parameter/value pairs, otherwise a list of such dictionaries in the order of the daemons were named when the interface object was constructed. The exact contents of a status dictionary is dependent on the version of the *memcached* serving the connection.

In case any of the daemons associated with the interface object cannot be contacted, an error results.

Example:

```
echo [dict get [memcache status $mchandle] uptime]
```

memcache stop

```
memcache stop mchandle
m.stop()
```

Attempt to stop all *memcached* daemons associated with an interface object by sending them termination signals. This works only with local daemons and requires that the effective user of the application has sufficient rights. It has no effect on any remote daemon process.

As a side effect, all active connections to servers are closed, but they are re-activated automatically if the daemons become available once more at a time when later commands are executed on the interface object. The command does not destroy or invalidate the interface object proper. Daemons can, for example, be restarted manually or by means of the **`memcache start`** command. However, restarted daemons have no memory of tuples stored previously, so a total data loss is always the consequence when a daemon was successfully terminated by means of this command, or died because of any other reason.

The return value of the command is 1 if all daemon processes could be terminated, 0 if there were any problems such as insufficient permissions or remote daemons, and -1 if no action was required.

memcache valid

```
memcache valid mchandle key
m.valid(key=)
```

A boolean check whether a given key is valid, i.e. whether data was stored with this key and has not expired. If the key is not valid, zero is returned, not an error.

Example:

```
if {![memcache valid $mch $mykey]} { ....
```

The *netcache* Command Extension

The *netcache* command extension provides an interface to the **NCBI** *netcache* memory data caching daemon. This is a useful functionality especially for the implementation of stateful **CGI** and **FCGI** applications.

Since the **NCBI** Toolkit, which provides the basic interface functions, is difficult to compile on many platforms, this command extension is only provided as part of the standard toolkit distributions for a few select platforms. Even these packages do not contain the *netcache* daemon proper, only the interface module. Setting up a **netcache** environment is probably not justified for most application scenarios, especially since the simpler and much more portable **memcache** command extension and associated daemon with near equivalent functionality are now a standard toolkit component.

There is currently no **PYTHON** interface to this extension.

This command extension is thread-safe.

These are the subcommands:

netcache create

```
netcache create clientname ?host? ?port? ?lbname?
```

Create a *netcache* daemon interface object. The command returns a handle for this object which is used by all other **netcache** commands to identify the interface. Multiple interfaces can be in use simultaneously.

The *clientname* argument is an arbitrary string which is used to identify an application, or application component. The optional *host* and *port* arguments specify the server the storage daemon is running on. If not set, they default to *localhost* and the default *netcache* port 9001. The final argument can be used to select a load balancer component. Please refer to the **NCBI** *netcache* documentation for details. The default is an empty string, i.e. the host/port combination directly selects the physical server.

Example:

```
set nhandle [netcache create "killerapp" $cachehost]
```

netcache delete

```
netcache delete all
netcache delete ?nhandle?...
```

The first command version deletes all active netcache interface objects, after properly closing their connections. The second variant closes and deletes specific handles. The command returns the number of deleted interface objects.

The functionality of the first command variant is automatically executed if the **netcache** command extension is unloaded.

Deleting an interface handle does not immediately delete the tuples created via this interface on the daemon. The data remains stored until the time-to-live expires, and may be fetched by other interface object instances, even in different processes on different hosts, if they connect to the same daemon, use the same client name, and supply the proper keys. To delete tuples, use the **netcache remove** command.

Example:

```
netcache delete $nh
```

netcache get

```
netcache get nhandle key ?listindex?
```

Retrieve the data associated with the specified key. The data is returned as the command result. In case there is no tuple associated with the key, including due to expiration of the lifetime of data associated with the key on the remote daemon, an error results. The returned data can be binary.

If the optional list index is specified, the data is expected to be a properly formed **Tcl** list. If interpretation as list succeeds, only the selected list element is returned. If the list index is larger than the number of list items in the data, an empty string is the result, but no error is raised.

Example:

```
set data [netcache get $nh $key]
```

netcache list

```
netcache list ?pattern?
```

Get a list of currently used *netcache* object handles. If no *netcache* objects have been created, an empty list is returned. Optionally, a string filter pattern can be specified.

Example:

```
netcache list
```

netcache put

```
netcache put nhandle data
netcache put nhandle key data ?ttl_secs?
```

Store data on a *netcache* daemon. The data is stored as an opaque byte array and can be of any type. It is not necessary to base64-encode the data or protect it in any other fashion.

The first command variant automatically assigns an unique key and stores the data with it.

The second variant is more flexible. If the key parameter is an empty string, a new unique key is automatically generated. However, with an explicit key, data can be replaced if the key is already in use. If an explicitly specified key does not exist, a new tuple is silently created. Keys are also byte arrays and could be binary, but commonly some readable string encoding is used. Optionally, the time-to-live of the data on the daemon can be specified. The default value is part of the netcache daemon configuration and cannot be queried directly. Tuples which are in the store longer than their allowed time are automatically retired by the daemon. Attempts to access them via their key after this point in time results in an error.

The command returns the key of the tuple, which may have been assigned automatically.

Example:

```
set key [netcache put $nh $data1]
netcache put $nh $key $data2
```

netcache remove

```
netcache remove nhandle key
```

Delete a tuple on the remote daemon. If the key is not valid, an error results.

This command does not delete the interface object. For this purpose, use the **`netcache destroy`** command.

Example:

```
catch {netcache remove $nh $key}
```

This statement removes the tuple, and ignores any errors in case the key is not stored or has timed out.

netcache valid

`netcache valid` *`nhandle key`*

A boolean check whether a given key is valid, i.e. whether data was stored with this key and has not timed out. If the key is not valid, zero is returned, not an error.

Example:

```
if {![netcache valid $nh $mykey]} { ....
```

The *pubchem* Command Extension

The **`pubchem`** command enables the toolkit to talk directly to the **NCBI PUBCHEM** database server. This means there is a direct database client connection, not an access to any of the public **APIs**. This command therefore only works in environments which have network access to the **NCBI** server farm, or an in-house clone.

This command is implemented as a **CACTVS** command extension, not a standard **TCL** module, because it links into **CACTVS**-specific internal data structures and cannot be loaded into a standard **TCL** interpreter.

The command extension can be explicitly loaded via a

```
cmdx load pubchem
```

The command is also auto-loaded in standard interpreters, if the command extension module can be found in the search path. It is a built-in command in some versions of the **`csweb`** **CGI** interpreter.

The command is currently not part of the standard toolkit distribution.

There is currently no **PYTHON** interface for this extension.

These are the subcommands of the extension:

pubchem cigs

```
pubchem cigs cidvalue ?type?
```

Get structure identity group information of a CID. If no type parameter is given, or it is given as *all*, the full set of CIGs is returned as a list in the order *tautomer*, *connectivity*, *stereo*, *isotope* and *exact*.

If a type parameter is specified, as one of the allowed values *all, tautomer, connectivity, isotope* and *exact* or its abbreviation *a, t, c, i* or *e*, only the selected group identifier is reported.

If the CID is not found in the database, an error is reported.

Example:

```
echo [pubchem cigs $cid t]
```

pubchem dump

```
pubchem dump ?updateonly? ?settimestamp? liveVar deadVar
```

This command is used to get information about changes in the main *PCCompound* database since the last query. By default, both the *updateonly* and *settimestamp* flags are unset. The command sets the two variables named in the arguments to a list of CID identifiers. If the *updateonly* flag is set, only CIDs which have changed (i.e. were added, modified, or deleted) since the last time stamp setting are reported. Records which are still in the database are stored in the live variable, deleted records are returned in the dead variable. If the *updateonly* flag is not set, all database identifiers are returned.

The *settimestamp* flag controls whether this query should automatically set the processing time stamp of the returned records to the current time, thus marking the database records as synchronized. Records with an updated time stamp are excluded from the result list of further **`pubchem dump`** commands with the *updateonly* flag set.

This command can take a couple of seconds to execute, and the result lists can be large (up to a million or more list elements).

pubchem fetchblob

```
pubchem fetchblob sid sidvalue
pubchem fetchblob cid cidvalue
```

This command retrieves a binary **ASN.1** structure record blob from the database. The command comes in two variants. Retrieval via the SID returned the complete structure record, with all embedded structure forms and their CIDs, while access via a CID only yields that structure and its property data.

The raw blob data is returned as result. If the queried SID or CID does not exist in the database, an error is raised.

For historical reasons, the command can also be used without the *sid* or *cid* access type identifier. This form is equivalent to access via an SID.

Example:

```
set blob [pubchem fetchblob cid 999]
filex load asnb
set eh [molfile read [molfile open $blob s]]
```

This command sequence creates an ensemble object from the **ASN.1** blob. Note that in most cases the **`pubchem fetchens`** command is more convenient to use for this purpose. Structure processing options should be completely disabled in order to avoid any change when reading compound data from raw blobs, as in

```
molfile set $fh readflags {}
```

pubchem fetchens

```
pubchem fetchens sid sidvalue
pubchem fetchens cid cidvalue
```

This command retrieves an ensemble from the **PubChem** database via an SID or CID identifier. The return value of the command is a new ensemble handle. In case an SID or CID is not found in the database, an error is raised.

If the retrieval is made via a CID, only that CID and its associated data is returned. For access via SID, the full content of the **ASN.1** record is encoded in the returned ensemble. The connectivity of the structure for which its handle is returned is that of the deposited structure. Standardized compounds and other structure variants of the deposited structure are attached to this base structure as one or more properties `E_NCBI_COMPOUND` of datatype *ensemble*. These secondary property-encoded ensembles store the data registered for them in the database in their own independent set of properties. In theory, this could include further structure derivatives that are again stored as properties `E_NCBI_COMPOUND`.

For historical reasons, the command can also be used without the *sid* or *cid* access type identifier. This form is equivalent to access via an SID.

All structure processing options are disabled when decoding the blob into the ensemble, so the returned structure is a faithful representation of the original data, including all bond types, bond annotations and charges.

Example:

```
set eh [pubchem fetchens sid 999]
set ehstd [ens show $eh E_NCBI_COMPOUND]
set stdcid [ens show $ehstd E_NCBI_COMPOUND_ID(id)]
```

This example retrieves a full **PubChem** record via an SID, isolates the first structure variant encoded in the record (which is the default standardized form), and then reads out from that standardized form its CID.

pubchem setdbhosts

`pubchem setdbhosts` *`hostlist`*

This command changes the default set of database cluster hosts from the compiled-in default. The *hostlist* parameter is a list of one or more host names in standard **Tcl** notation.

Example:

```
pubchem setdbhosts [list DDDSQL10 DDDSQL11]
```

pubchem sidlist

`pubchem sidlist` *`cidlist`*

This command returns a nested list of SIDs associated with CIDs. For each CID in the *cidlist* parameter a list element is returned which contains the list of associated SIDs.

Example:

```
set cidlist [list 1 2 3]
set sidlist [pubchem sidlist $cidlist]
foreach cid $cidlist sidset $sidlist {
   puts "CID $cid is associated with [llength $sidset] SIDs"
}
```

pubchem sids

`pubchem sids` *`cid`*

This command returns a list of all SIDs a CID is associated with. An error is raised if the CID is not found int the database.

pubchem synonyms

`pubchem synonyms sid` *`sidvalue`*
`pubchem synonyms cid` *`cidvalue`*

Get the list of synonyms associated with a SID or CID. The command returns a string list. In case there are no synonyms, or the identifier is not found in the database, an error is returned.

The synonyms list contains only names registered in the global synonyms data block of the **ASN.1** specification. It does not report any additional names which may be stored in property data areas of individual compounds of the record.

pubchem subcommands

`pubchem subcommands`

Return a list of the subcommands of the **`pubchem`** command.

The *stat* Command Extension

Auxiliary Tcl and Python Commands

Beside the listed commands for manipulating chemical and non-chemical objects, the standard **Tcl** scripting interface is enhanced with a collection of additional commands which do not belong to one of the two big groups. This is a list of these commands:

- *alarm*
  Access to the standard `alarm()` C library function. Only relevant for the Windows version, because on Unix the equivalent *alarm* command provided by the **TclX** library is available.
- *avg*
  Compute average of arguments
- *bitvector*
  Operate on string or optimized internal representations of bit vectors
- *bread*
  Read binary data
- *bwrite*
  Write binary data
- *color*
  Decode color specifications
- *creverse*
  Reverse string
- *daemonize*
  Convert normal process into demon process (Unix versions only)
- *decode*
  Decode data in various encoding and compression schemes
- *encode*
  Encode data in various encoding and compression schemes
- *fcgi*
  Perform I/O as a **FastCGI** application
- *fetch*
  Retrieve data via **URL**s
- *filecheck*
  Check file formats
- *insidepolygon*
  Check whether a point is within a polygon.
- *ldelete*
  Delete selected elements from a **Tcl** list
- *lineintersect*
  Check whether line segments overlap

- *lreverse*
  Reverse TCL list
- lsearch
  Extended features vs. the standard TCL version
- *lsum*
  Compute sum over list elements
- *lvardelete*
  Delete selected elements from a TCL list variable
- *mailcap*
  get standard opener/viewer for **MIME**-typed files
- *map*
  systematically apply command to list elements
- *mimetype*
  get MIME type from file extension
- *parse*
  verify decodability of strings representing of certain data types
- *passwd*
  operations on standard Unix-style passwords
- *post*
  simulate the posting of a HTML form
- *prod*
  compute product of arguments.
- *python*
  execute commands in a Python interpreter
- *quote*
  properly quote strings
- *random*
  generate thread-local random numbers
- *rpc*
  get information about rpc services (Unix only)
- *screen*
  perform structure fragment screening operations with bitvectors
- *sqsum*
  compute sum of squared arguments
- *stddev*
  compute standard deviation of arguments
- *sum*
  compute sum of arguments

- *tmpdir*
  get current temporary directory
- *tmpfile*
  get **Tcl** file handle of file opened for reading and writing in temporary directory.
- *tmpname*
  obtain name of a new *tmp* file
- *uncgi*
  decode WWW form data for **CGI** applications
- *unzip*
  extract data from nested lists
- *vec*
  basic vector operations
- *zip*
  merge lists in an interleaved fashion

Mathematical expression enhancements

The **Cactvs** toolkit adds a number of commonly used mathematical functions to the interpreters. For **Tcl**, this means they are implemented as extensions to the math expression engine. The mathematics enhancements are implemented as normal module functions in **Python**. **Python** does not distinguish between normal and math functions.

The following functions are available:

- *apos(a)*
  Normalize radian angle to 0..2π
- *astd(a)*
  Normalize radian angle to range -π...π
- *bitrange(low,high)*
  Return a wide integer with all bits set between the specified positions, with the lowest bit position starting as zero, and inclusive boundaries. If the *low* position is specified as a negative number, the start position is the lowest bit (zero). If the *high* position is specified as a negative number, the stop position is the highest bit in use in a wide integer, which may be platform-dependent but usually is 63 for 64-bit wide integers. If both positions are specified as negative numbers, the result is zero and not a word with all bits set.
- *clamp(x,low,high)*
  Restrict x to minimum *low* and maximum *high.* This is an alias for the *limit* function.
- *cmp(x,y)*
  Returns 0 if the values are equal, -1 if *y* is larger than *x*, or 1 otherwise.
- *deg(x)*
  Convert radians to degrees

- *ffs(x)*
  Get the position of the first bit set in the input value interpreted as unsigned wide integer. Bit position numbering begins with one. If no bits are set, zero is returned.
- *fls(x)*
  Get the position of the last bit set in the input value interpreted as unsigned wide integer. Bit position numbering begins with one. If no bits are set, zero is returned.
- *gfraction(minvalue,maxvalue,nticks)*
  Suggest a nice value for the number of fractional digits for the numeric tick labels of a graph display of data where the values for the axis lie between the minimum and maximum arguments, and the axis has approximately the specified number of ticks.
- *gmax(minvalue,maxvalue,nticks)*
  Suggest a nicely rounded maximum value for a graph display of data where the values for the axis lie between the minimum and maximum arguments, and the axis has approximately the specified number of ticks.
- *gmin(minvalue,maxvalue,nticks)*
  Suggest a nicely rounded minimum value for a graph display of data where the values for the axis lie between the minimum and maximum arguments, and the axis has approximately the specified number of ticks.
- *gstep(minvalue,maxvalue,nticks)*
  Suggest a suggest nicely rounded tick step value for a graph display of data where the values for the axis lie between the minimum and maximum argument, and the axis has approximately the specified number of ticks.
- *htonl(x)*
  Get integer value in network byte order.
- *hue2rgb(p,q,t)*
  A helper function for color conversion, see *https://stackoverflow.com/questions/2353211/hsl-to-rgb-color-conversion*
- *isign(x)*
  Get the sign of the argument as integer value (-1, 0 or 1).
- *limit(x,low,high)*
  Restrict x to minimum *low* and maximum *high*.This is an alias for the *clamp* function.
- *log10(x)*
  Get decadic logarithm.
- *max(x,y)*
  Get the maximum of the two input values. The result type depends on the input data and automatically uses the required type to represent the result in its full precision. This function is not implemented in **PYTHON** because it is present right out of the box.
- *min(x,y)*
  Get the minimum of the two input values. The result type depends on the input data and automatically uses the required type to represent the result in its full precision. This function is not implemented in **PYTHON** because it is present right out of the box.

- *mod(x,y)*
  Compute *x modulo y*. This function can process floating point values, whereas the standard TCL % operator can not.
- *nice(x,r)*
  Get a nice close number of *x* (i.e. something based on a power of 1,2,5,10,...), either by rounding up (*r* is not null) or rounding down (*r* is null).
- *ntohl(x)*
  Get integer value in host byte order
- *pi()*
  Returns π with full precision.
- *pix2pt(x)*
  Convert the pixel count argument to a points value, assuming a resolution of 72 *dpi*.
- *pt2pix(x)*
  Convert the typographical points argument to a pixel count value, assuming a resolution of 72 *dpi*.
- *rad(x)*
  Convert degrees to radians.
- *rnd(x)*
  Get an integer random number between zero and the integer parameter minus one. This function is integrated with the *random* command extension and is affected if the random generator state is modified via this command. Every interpreter thread (but not slave interpreter) has its own independent random number generator.
- *range(x,low,high)*
  Check whether value is within range. The result is either boolean 0 or 1. To avoid name collision with the standard `range()` function, the PYTHON version uses the function name *inrange*.
- *sqr(x)*
  Square value of *x*.
- *zero(x)*
  Check whether value is close to zero. Usually, this means the argument is within +/- .00001 to zero, but this is configurable.

avg

```
avg arg ...
avg(arg,...)
```

Interpret the arguments as numeric values and compute their average.

bitvector

```
bitvector and vector ?vector?...
bitvector cmp vector1 vector2
bitvector cmpall vector1 vector2
bitvector count0 vector1 ?vector2?...
bitvector count1 vector1 ?vector2?...
bitvector create length ?basevalue? ?togglepositions?
bitvector distance vector1 vector2
bitvector or vector ?vector?...
bitvector nand vector ?vector?...
bitvector nor vector ?vector?...
bitvector not vector
bitvector parse vector
bitvector record0 vector ?offset? ?length?
bitvector record1 vector ?offset? ?length?
bitvector screen vector1 vector2
bitvector screenall vector1 vector2
bitvector size vector1 ?vector2?...
bitvector test0 vector ?offset? ?lengt?
bitvector test1 vector ?offset? ?lengt?
bitvector xor vector ?vector?...
Bitvector(length=,?basevalue=?,?togglepositions=?)
Bitvector.Create(length=,?basevalue=?,?togglepositions=?)
Bitvector.Parse(data)
bv.cmp(vector)
bv.cmpall(vector)
bv.compare(vector,?cmpflags?,?c1?,?c2?)
bv.count0()
bv.count1()
bv.distance(vector)
bv.m_and(?vector=?,...)
bv.m_not()
bv.m_or(?vector=?,...)
bv.nand(?vector=?,...)
bv.nor(?vector=?,...)
bv.record0(?offset=?,?length=?)
bv.record1(?offset=?,?length=?)
bv.screen(vector)
bv.screenall(vector)
Bitvector.Size(?vector?,...)
bv.test0(?offset=?,?length=?)
bv.test1(?offset=?,?length=?)
bv.xor(?vector=?,...)
```

This command provides basic functionality for the processing of bit vectors on a string level, or as binary pre-parsed objects. String representations of bit vectors understood by this command consist of 0 and 1 characters, optionally prefixed by a percent or *B* character. Bitvectors with this encoding style can, for example, be obtained from a **Tcl** data recall command on chemistry object bitvector data items.

In both the **Tcl** and **Python** interfaces, decoded bitvectors are internally represented as binary custom language interface objects. This avoids re-parsing strings if the objects are used in multiple processing commands.

Example:

```
set vector [ens get $ehandle E_SCREEN]
```

The following subcommands are available:

`bitvector and` *`vector ?vector?...`*

`bitvector` *`or`* ***`vector ?vector?...`***

`bitvector xor` *`vector ?vector?...`*

`bitvector nand` *`vector ?vector?...`*

`bitvector nor` *`vector ?vector?...`*

These are the standard boolean operations. *nand* and *nor* yields the inverted result of the *and* and *or* operations. If no arguments beyond the first are given, the result is the input vector for *and, or, xor*, and the inverted input vector for the rest. In case the vectors are of different lengths, they are virtually padded with zero bits to the size of the largest vector. *eor* is an alias name for the *xor* operation.

`bitvector not` *`vector`*

Invert the vector in bit-wise fashion.

`bitvector test1` *`vector ?offset? ?len?`*

`bitvector test0` *`vector ?offset? ?len?`*

Return a list of the vector index positions (starting with zero) which have a set or unset bit. Optionally, an offset for the first bit to be tested, and the maximum number of bits to be tested may be specified. By default, testing begins with the first vector position (index 0) and continues until the end of the vector. The subcommand *test1* may be abbreviated to *test*. **`bitvector index0`** and **`bitvector index1`** are aliases for this command.

`bitvector record1` *`vector ?offset? ?len?`*

`bitvector record0` *`vector ?offset? ?len?`*

These commands are equivalent to the **`test0/index0`** and **`test1/index1`** commands above, except that they report the selected bit positions starting with one instead of zero, in the customary fashion of file record counting.

`bitvector create` *`length ?basevalue? ?togglepositions?`*

Generated a new bitvector of the specified length. By default, it is set to all zeros. The base value parameter, which may be 0 or 1, can be used to generate a vector with all set bits instead. If the toggle positions argument is used, it is expected to be an integer list with 0-based vector indices were the bit value in the new vector should be flipped.

`bitvector parse vector`

This command parses a vector representation (which can be a string representation, or already a bitvector object, in which case no real work is performed) and creates a new pre-parsed bitvector object.

`bitvector screen` *`vector1 vector2`*

Perform a screening operation on the first bitvector. The command returns the first 0-based index position where a bit is set in the second vector, but not in the first. If all set bits in the second vector have counterparts in the first vector, minus one is returned. The bitvectors may be of different length. The shorter vector is assumed to be filled up with zero bits. In a standard application, the first argument is a feature vector of a structure, and the second vector a feature vector of a substructure.

`bitvector screenall` *`vector1 vector2`*

Perform a screening operation on the first bitvector. The command returns a list of all 0-based bit positions where a bit is set in the second vector, but not in the first. If all set bits in the second vector have counterparts in the first vector, an empty list is returned. The bitvectors may be of different length. The shorter vector is assumed to be filled up with zero bits. In a standard application, the first argument is a feature vector of a structure, and the second vector a feature vector of a substructure.

`bitvector count1` *`vector1 ?vector2?...`*

`bitvector count0` *`vector1 ?vector2?...`*

Count the total number of set or unset bits in all the specified vectors. **`bitvector count`** is an alias for **`bitvector count1`**.

`bitvector distance` *`vector1 vector2`*

Compute the Hamming distance between the vectors.

`bitvector cmp` *`vector1 vector2`*

Compare the two bitvectors and return the 1-based position of the first bit which disagrees. If the bit is set in the first vector, a the position is returned as a positive number, and as a negative number when the bit is set in the second vector. If both vectors are identical, or are identical if the shorter vector is assumed to be filled up with zero bits, the return value is zero. The alternative subcommand name **`compare`** is an alias.

`bitvector cmpall` *`vector1 vector2`*

Compare the two bitvectors and return a list of the 1-based bit positions where the two vectors disagree. If at any such position the first vector has a set bit, the position is listed as a positive number. In the opposite case, the position is entered as a negative number. If the vectors are identical, or are identical if the shorter vector is assumed to be filled up with zero bits, the return value is an empty list. The alternative subcommand name **`compareall`** is an alias.

`bitvector size` *`vector1 ?vector2?...`*

Returns the size of the largest bit vector in the argument set.

The **Python** implementation additionally supports standard copy and pickle methods, *with*-clause, and overloaded numeric operators **~**,**&**, **^**, **|** as well as **&=**, **|=**, **^=**. The **`len()`** method returns the vector length. A bitvector has a boolean true value if any of its bits are set. Setting and retrieving individual bits via indices is also supported:

```
b[0] = b[1]
```

Due to syntactic limitations in the generic **Python** parser, the **`and`**, **`or`** and **`not`** methods cannot be named such - these are reserved words. Instead, they are accessible as **`m_and`**, **`m_or`** and **`m_not`**. The other bit methods (**`nand`**, **`nor`**, **`xor`**) are implemented with their normal name, but also accessible with the same prefix. The **Python**-only method **`compare`** is equivalent to a **`prop compare`** statement, except that it is an instance method, not a class method.

bread

```
bread tcl_filehandle format ?var..?
```

This command reads formatted binary data from an output channel. The specified **Tcl** channel is automatically configured for binary data for the duration of the command and then restored to the original state.

The command is complementary to the **bwrite** command described below and uses the same formatting specifications.

Multiple values sharing the same format can be read in one statement with a set of recipient variables is specified. The return value is the value of the last item read. If no variables are used, one item is read and returned, but not stored in a variable.

This is a **Tcl**-only command.

Example;

```
set s [bread $channel string16]
```

bwrite

```
bwrite tcl_filehandle format data ?data?...
```

This command writes formatted binary data to an output channel. The specified **Tcl** channel is automatically configured for binary data for the duration of the command and then restored to the original state.

This is a **Tcl**-only command.

If more than one data item is specified, the same format is used for all the data items. For most formats, the binary data layout is not changed and it thus platform-dependent. The exception are those formats which are prefixed with an X: These follow the platform-independent **XDR** encoding standard (RFC 1014) in their layout (network byte order, MSB first), but **not** in the stored item size (the byte size of smaller objects is not expanded to multiples of four). The following formats are supported:

- *long, int*
  4-byte signed integer
- *ulong, uint*
  4-byte unsigned integer
- *xlong, xint*
  4-byte signed integer, MSB layout
- *xulong, xuint*
  4-byte unsigned integer, MSB layout
- *short*
  2-byte signed integer
- *ushort*
  2-byte unsigned integer

- *xshort*
  2-byte signed integer, MSB layout
- *xushort*
  2-byte unsigned integer, MSB layout
- *char, xchar*
  1-byte character
- *ulong8*
  8-byte unsigned integer
- *xulong8*
  8-byte unsigned integer, MSB layout
- *float*
  4-byte floating point number
- *xfloat*
  4-byte floating point number, MSB layout
- *double*
  8-byte floating point number
- *xdouble*
  8-byte floating point number, MSB layout
- *string*
  character string. If no length is specified, it is terminated by a zero byte.

The string format may contain additional length and pad character specifications. If a pad character is used, it must be supplied as its ISO Latin code. This makes it simple to use zero bytes as filler.

Example:

```
bwrite stdout string16:[ctype ord x]
```

writes a string which always occupies exactly 16 bytes. If it is longer, the extra characters are ignored. If it is shorter, it is padded with “x“ characters. A zero byte is **not** written, because this string has an explicit length.

The **`bread`** command provides the complementary functionality to read binary data into variables using the same format specifications as template.

cleanup

```
cleanup
cleanup()
```

A utility cleanup command which closes (where applicable) and removes all deletable chemistry objects with valid handles. Removed object classes include **`ens`**, **`reaction`**, **`dataset`**, **`molfile`**, **`network`**, **`table`**, **`biologics`**, **`hierarchy`**, and **`lhasa`** engines. The return value is the number of deleted objects.

color

```
color ?-alpha? ?-depth n? ?-hex? ?-javaint ?-name? ?-shade delta? ?-tuple?
colorname/colorspec
```

```
color(color=,?format=?,?depth=?,?alpha=?,?shade=?)
```

Decode a color name. On Unix systems, color names are looked up in the local **X11** color database. On the **PC** platform, a representative **X11** color database dump is compiled into the application. In addition to English color names, the standard hex color notation, such as *#rrggbb* or *#rgb*, may be used, with or without alpha channel data. If a hex color notation has 4, 8 or 16 hex digits, the value is interpreted as *#rgba*, *#rrggbbaa* and *#rrrrggggbbbbaaaa*, respectively. In case of 12 hex digits (12 is both divisible by 3 and 4), the interpretation *#rrrrggggbbbb* takes precedence. Color names are case-insensitive.

By default, a color depth of 16 bits is assumed, and the returned color component values are thus in the range 0..65535. Smaller or larger color component value ranges may be specified by an explicit *depth* value, which must be in the range between 2 and 24. Depth 8 is used for the commonly used format with 8 bits per channel.

If the -*shade* option is used, the decoded color value is darkened or brightened by the specified amount by component-wise addition before it is output in the selected format. The shading value is scaled according to the selected color depth, i.e. with a color depth of 8, the useful minimum and maximum scale values are -255 and +255, while with a color depth of 16 the limits are -65525 and +65535. The transformed color values are automatically clamped to the white and black extremes, so the output will always be a valid color representation. Shading is not applied to the alpha/opacity channel. A shading value of zero has no effect in any color depth.

The default return format is a list with the decimal RGB values of the decoded color. This corresponds to the explicit mode -*tuple*. If the -*hex* option is set, the output is formatted as a single hex-encoded color value, and -*javaint* returns a standard **Java** color signed int color value. If the -*alpha* option is set, the format of the output includes the opacity value as the fourth component in RGBA order, either as an additional list element or appended to the hex string. The -*javaint* value always includes an alpha component.

With the -*name* option the command attempts to find the most closely matching color name in the database for the decoded and transformed color values and return that name instead of a color component list or hex encoding.

The **Python** version uses a single format argument. It can be set to either *tuple*, *hex*, *name* or *javaint*.

The command may also be spelled **`colour`** instead of **`color`**.

Example:

```
color -depth 8 -hex Red
```

returns **`#FF0000`**.

```
color -depth 8 -shade -0x10 -hex #808080
```

returns **`#707070`**

creverse

```
creverse string
```

```
creverse(string)
```

Reverse a string.

Example:

```
creverse abc
```

returns *cba*.

daemonize

```
daemonize ?priority? ?closefiles?
daemonize(?priority=?,?closefiles=?)
```

Transform the current process into a daemon process which is decoupled from all control terminals and runs in the background until finished or terminated. The current process is forked, and the old foreground process exited.

By default, the background process priority is unchanged. Alternatively, a new priority may be specified as the first optional argument. If it not an empty string, an attempt is made to set the process priority to the new value. No error message is generated when the attempt fails. Useful priority values depend on the platform. On Linux, the range is 0...20, with 20 being the *lowest* priority. Increasing the priority, by using a value *lower* than the current process value, requires non-standard permissions, usually an effective application user ID of *root*.

By default, all open file handles are closed. If the optional *closefiles* parameter is set to 0, most file handles are kept open. An exception are the standard input, output and error channels. These are always redirected to */dev/null,* regardless of the option value. Note that the closing of the file descriptors does not automatically invalidate any **Tcl** scripting language references to these, such as standard **Tcl** file or socket handles, or toolkit **molfile** handles. These should be explicitly closed by the application script before this function is called. Otherwise, any use of these stale handles results in errors.

This command is only available on Unix-based platforms.

decode

```
decode mode data ?extra?
decode(mode=,data=,?extra=?)
```

This command decodes a number of commonly encountered encoding formats for string and binary data. The decoded data is returned as command result. The mode parameter decides which decoding scheme is used. For the **Python** version, the mode is usually specified without the leading -, but this is not required. The following modes are available:

- *-base64*
  The data argument is expected to be a standard base64 encoding or the **URL** variant thereof (with - and _ replacing the + and = characters). The data characters may be either written as one unbroken sequence or follow RFC2045, which mandates embedded line breaks and an encoding data characters count divisible by four. The appropriate subformat is automatically detected. The decoded data may be binary.

- *-bitset*
  Decode a collection of symbolic bit names via a standard toolkit bitset enumeration value specification, as it is used for example in property definitions. The enumeration definition syntax is documented in the paragraph dealing with the *enum* property attribute. This command variant uses the syntax
  ```
  decode -bitset symvalue enum_spec
  ```
  The return value is a wide integer value with all bits set according to the decoded symbolic value string. If the value cannot be decoded, an error results. An empty value argument always yields zero as result. Example:
  ```
  decode -bitset a|c none:a,alpha:b,beta:c:d
  ```
  The return value is 5, which is combined from set bit index 0 for *a* and set bit index 2 for *c*. For bitsets, the first colon-separated word in the enumeration specification contains one or more aliases for zero set bits, which corresponds to the numerical result zero. Instead of a vertical bar, whitespace may also be used as bit position name separators.

- *-datauri*
  Decode the payload of a data **URI** in base64 or **URL** encoding. The **MIME** type and character set information are not used and ignored if present. The **URI** may encode binary data.

- *-dataurl*
  An alias for *-datauri*

- *-enum*
  Decode a symbolic name via a standard toolkit enumeration value specification, as it is used for example in property definitions. The enumeration definition syntax is documented in the paragraph dealing with the *enum* property attribute. This command variant uses the syntax
  ```
  decode -enum symvalue enum_spec
  ```
  The return value is the integer value linked to the symbolic value argument. If the value cannot be decoded, an error results. Example:
  ```
  decode -enum b a,alpa:b,beta:c=99
  ```
  The returned value is 1, which is the default value of symbolic name *b*, or its alias *beta*, taken from its colon-separated word index. The word index is used in the absence of an explicit value, as it is specified in the example for value *c*.

- *-formula*
  The input is expected to be an Unicode string representing a molecular formula in standard notation with element counts encoded in subscript digits, and charges encoded with superscript digits or plus/minus characters. The output is the plain ISO-8859-1 string form of the same formula.

- *-gzip*
  Because the *zlib* and *gzip* formats are automatically distinguished, this is an alias for *-zlib*.

- *-gzip64*
  Because the *zlib* and *gzip* formats are automatically distinguished, this is an alias for *-zlib64*.

- *-hex*
  The data is expected to be a sequence of hex digit pairs, specifying the **ISO** Latin1 value of the decoded character at every position. White space between hex characters is ignored, but other non-hex characters raise an error.
- *-html*
  The input data is expected to be **HTML** text with entity encodings, such as *&quot;* for a quotation mark. These entity encodings are resolved to a standard **ISO** Latin encoding. Note that there are entities which cannot be represented in the **ISO** Latin encoding. These are passed unchanged. This mode also decodes **XML** encoding, which uses a smaller set of entities. Neither **HTML** nor **XML** decoding removes tags.
- *-json*
  Decode a **JSON** object into an equivalent **TCL** list (for array elements) and dictionary (for map elements) representation. This option is only available if the interpreter was compiled with **JSON** support.
- *-loinc*
  Decode a **LOINC** ID (`https://loinc.org/`) and get the textual description of the data content. The first call to the **LOINC** decoder takes a few seconds since the **LOINC** table needs to be loaded from disk.
- *-mysql*
  Decode a binary blob as produced by the **MYSQL** database function `compress()`. This is primarily useful in the context of the **MYSQL** database cartridge. The data consists of a 4-byte binary header with the decompressed data length, followed by zlib-compressed data. An empty string is encoded verbatim.
- *-paper*
  Decode a paper size such as *A4* or *letter*. If the paper size is known, return a tuple with the width and height of the format, in that order. If the format is unknown, an error results.
- *-pdf*
  Decode input as a **PDF** string. If the first character is an opening parenthesis, and the last character a closing parenthesis, these are also stripped.
- *-python*
  Decode a standard **PYTHON** string object representation, for example a quoted string, a simple number, or a tuple, list or dictionary. The result is a suitable **TCL** interpreter value object containing the equivalent information.
- *-quoted-printable*
  The input is expected to use quoted-printable representation, such as *=20* for a space character. The output is in ISO Latin1.
- *-rc4*
  The input data are bytes encoded with an **RC4** cipher. This command version requires an additional argument after the data argument which is used as the decoder key.
- *-rfc2045*
  This is the same as *-base64*.

- *-robustzip*
  Slower than *-zip*, but more robust against extreme compression factors and data corruption.
- *-rtf*
  Strip formatting information from **RTF** text. Only the plain text parts, encoded as ISO Latin1, remain. Leading and trailing white space is removed. The only formatting instruction which leaves a trace in the decoded text is *\par* (paragraph break), which is translated into a space character. The decoder implements a minimal **RTF** parser, so that any legal **RTF** string can be submitted, regardless of the specific formatting instructions it contains.
- *-suffix*
  Get the name of the most common chemistry structure or reaction file format using the suffix of a passed file name argument. If this is not a known suffix, the return value is *unidentified*. The returned name is suitable for use with the **filex** command. The argument file is not required to exist - it is only used to extract the suffix. This command does not try to match table or network file extensions, but it ignores known compression suffixes such as *.gz*, *.Z* or *.bz* in determining the applicable core file suffix.
- *-time*
  Try to decode a time or date value using the standard set of internal patterns and default mechanisms. The return value in case of success is the number of seconds since epoch time. Non-obvious time formats should be decoded by the scripting-language specific parsers (for example, **clock scan** for **TCL**, **datetime.strptime** for **PYTHON**)
- *-url*
  The input data is expected to be in **URL** encoding, with + for space and/or *%xx* (such as *%20* for space) hexadecimal three-letter encodings for other letters. These encodings are resolved to **ISO** Latin1.
- *-urlvbar*
  The input data is expected to be in **URL** encoding, with + for space and/or *%xx* (such as *%20* for space) hexadecimal three-letter encodings for other letters. These encodings are resolved to **ISO** Latin1. Vertical bar characters are translated to the system default line end character(s).
- *-zlib*
  The input data is expected to be *zlib*-compressed. Both direct *zlib* compression and string images of *gzip*-compressed files (which use a different header) are accepted. The subformat is automatically recognized. *gzip* header information, such as comments and the original file name, will not be preserved.
- *-zlib64*
  This is a convenient combination of *base64* decoding and *zlib* decompression. It is equivalent to first performing a *-base64* decoding, followed by a *-zlib* step.

The decoded data may contain zero-bytes and other special characters and may thus need special care in further processing. The counterpart of the **decode** command is the **encode** command, which can be used to encode data in all recognized decoder formats.

For historical reasons, the *-zlib** variants can also be invoked as the same command starting with *-zip**.

encode

`encode` *`mode data ?mode_arg?`*

`encode(mode=,data=,?extra=?)`

This command is the counterpart to the **`decode`** command. It is not completely identical, though . There are encoding schemes for which no corresponding decoding method has been implemented. Additionally, the decoder automatically recognizes certain variants of encoding schemes, which need to be selected explicitly on the encoder side. The encoded data, which may be binary, is returned as command result. In the **Python** version , the mode is usually specified without a leading -, but this is not required. The recognized encoding modes are:

- *-alertpage*
  Encode a **HTTP** response with header (but without status code) and an empty **HTML** body which opens up an alert box with the message when it is loaded. This is useful if the output is directed to an invisible Web frame.

- *-base64*
  Encode in *base64* format. The result string contains only printable characters. All data is packed into a long string without any line breaks. For line breaks after a maximum of 76 characters, use RFC2045 encoding.

- *-base64url*
  This is the same as *-base64*, except that the (not-quite-standardized, but frequently encountered) **URL**-safe encoding variant is used, which uses the '-' and '_' characters as replacements for '+' and '=' which need to be escaped in **URL**s. This is not the same as the more portable **URL**-encoding of the original base64 string, which would replace these characters by escaped notations.

- *-bitset*
  Encode an integer value to a combination of symbolic bit position names with a standard toolkit bitset enumeration specification, as it is for example used in property definitions. The syntax of this specification is explained in the paragraph on the *enum* property attribute. The command requires an extra mode argument for the enumeration definition string. In case a set bit in the numeric argument is not covered by the enumeration string, no error is raised. Instead, the numerical value of the bit is inserted. Example:

  ```
  encode -enum 3 a:b:c
  ```

  returns 'b|c'.

- *-boolean*
  Decode a boolean value in any of the forms recognized by **Tcl** and return it as *true* or *false* string value.

- *-boolobj*
  Decode a boolean value in any of the forms recognized by **Tcl** and return is as a native Tcl boolean result object. Normally, alternative boolean encodings such as integers or parseable strings are automatically decoded in scripts. An exception is when a function, such as the **JSON** encoder, needs to distinguish between, for example, an integer or boolean value, for proper execution of its task. In such context, an explicit object encoding can be useful.

- *-countryalpha2*
  Get the 2-character country code as per **ISO3166** from a recognized country name, or its 2-letter/3-letter/numeric ID.
- *-countryalpha3*
  Get the 3-character country code as per **ISO3166** from a recognized country name, or its 2-letter/3-letter/numeric ID.
- *-countryname*
  Get the official country name as per **ISO3166** from a 2-character, 3-character or numeric country code. It is also possible to search by full country name to verify its validity.
- *-countrynumeric*
  Get the numeric country code as per **ISO3166** from a recognized country name, or its 2-letter/3-letter/numeric ID.
- *-csource*
  Encode the data bytes as a comma-separated, hex-encoded byte sequence suitable for inclusion in C source files.
- *-cstring*
  Encode special characters such as *tab*, *vtab*, *cr*, *lf* as their C-style backslash encodings.
- *-crc32*
  Compute the **CRC32** checksum of the input data.
- *-datatag*
  Output a complete **HTML** tag (**`<img>`** or **`<object>`**) with an embedded data **URI** encoding of the data argument with an appropriate **MIME** type and the data encoded in *base64* format. This is useful for the generation of **HTML** pages with images where the images need not to be stored in external files. The encoded data must be a **GIF**, **PNG**, **JPEG**, **SVG** or **PDF** image, which is verified by looking at the magical header bytes. **PDF** input results in an **`<object>`** tag, the other types yield **`<img>`**. This command variant accepts an optional third argument. If it is set, it is used as the content of the “**`alt`**” tag attribute. Example:

```
prop set E_GIF datatype blob
encode -datatag [ens get $eh E_GIF] „Image of [ens get $eh E_IDENT]“
```

- *-datauri*
  Create a data **URI** in the format *data:mimetype;base64,payload* from the input data in base64 encoding. The default **MIME** type is *application/octet-stream*, but this can be overridden by using the optional extra parameter. No attempt is made to automatically determine the **MIME** type of the input.
- *-dataurl*
  This is an alias for *-datauri*.
- *-enum*
  Encode an integer value to a symbolic name with a standard toolkit enumeration specification, as it is for example used in property definitions. The syntax of this specification is explained in the paragraph on the *enum* property attribute. For this mode, extra mode argument is required and is the enumeration definition string. In case the numeric argument value is not covered by the enumeration string, no error is raised. Instead, the return value is the numeric input. Example:

```
encode -enum 2 a:b,beta:c,gamma
```

returns ‘c’.

- *-formula*
  The input is a plain ISO-8859-1 string representing a molecular formula in standard notation. The output is the same string, but with element count subscripts and charge superscripts encoded as Unicode sub/superscript characters in **UTF8** encoding.
- *-gzip*
  Encode in the format of the *gzip* program.
- *-gzip64*
  Encode in the format of the *gzip* program, and then encode the result in the *-base64* format.
- *-gzip64url*
  Perform *gzip* compression first, then encode the result in the *-base64* format, and finally perform **URL** encoding of the result, which is required because base64 can contain the characters +, = and /, which need to be escaped for **URL** contexts.
- *-hex*
  Encode by representing every input byte by a pair of hex digits.
- *-hexbytes*
  Encode by representing every input byte by a pair of hex digits, with a space between digit pairs.
- *-html*
  Encode for **HTML** display, using entities like *&quot;* for certain characters such as the quotation mark which can interfere with the tag structure.
- *-html0*
  Encode for **HTML** display, using entities like *&quot;* for certain characters such as the quotation mark which can interfere with the tag structure. This command variant however passes zero bytes verbatim, instead of encoding them as *&#0;*. Use of this command variant is discouraged. It it a hack which can be used to pass 0 bytes in simulated file uploads on certain browsers.
- *-htmlmultiline*
  Encode for **HTML** display, as with the *-html* mode, but encode line breaks as *<br>* tags.
- *-htmlpage*
  Encode the data string as a minimal but properly formatted complete **HTML** page, including **HTTP** headers (but no status code). This is primarily useful for for **CGI** or **FCGI** scripts to display a debug browser message.
- *-httpheader*
  Prepare a **HTTP** header string without the status code. The arguments are the content type and, optionally, the content length. This is primarily useful for **CGI** or **FCGI** scripts.
- *-imgtag*
  An alias for *-datatag*. Despite the name, this also covers auto-generated `<object>` and `<embed>` tags.

- *-jsarray*
  Encode the argument, which must be a properly formed **Tcl** list or **Python** tuple/sequence, as a **JavaScript** array initialization string suitable for inclusion into program-generated **JavaScript** code.

- *-jsdict*
  Encode the argument, which must be a properly formed **Tcl** or **Python** dictionary, as a **JavaScript** named object attribute initialization string suitable for inclusion into program-generated **JavaScript** code.

- *-jsnative*
  Encode the argument in the **JavaScript** type corresponding most closely to the **Tcl** or **Python** argument object type. This is not foolproof in **Tcl** since a **Tcl** object may temporarily be represented as string, depending on how it was set up and which operations it was subjected to, even if it could be represented as a more efficient type. However, usually it works, and the **Tcl** objects returned as the result of toolkit commands are of a suitable optimized type, not generic **Tcl** strings.

- *-jsonstatus*
  Prepare an **EXTJS JavaScript** toolkit submission response status, complete with **HTTP** header. The arguments are a boolean status value (1 = success, 0 = failure) and, optionally, a dictionary of error condition and error message values, or a dictionary of auxiliary result data. With a *false* status, the error dictionary is encoded under the e*rrors:* key in the response. For success, the auxiliary response data in encoded under the *data:* key.

- *-jsstring*
  Encode argument as a single-quoted **JavaScript** string suitable for inclusion into program-generated **JavaScript** code.

- *-quoted-printable*
  Encode using the *quoted printable* standard.

- *-qrcode*
  Return a **QR** code as **PNG** data **URL** string encoding the argument value. Usually the message content is a **URL**.

- *-md5*
  Compute an **MD5** hash code from the input string.

- *-pdf*
  Encode input data as a syntactically correct **PDF** string. The outer parenthesis pair is added automatically.

- *-python*
  Encode a **Tcl** interpreter value object, such as an integer, string, list or dictionary, into the standard string representation of the equivalent **Python** object. The proper operation of this command depends on the **Tcl** argument having the proper type. Since the command does not know the expected type, care must be taken when passing arguments - no automatic **Tcl**-style conversion, for example from a string to a list or dictionary, is performed. These operations must be explicitly coded if needed. This encoding scheme is not supported in **Python**.

- *-rc4*
  Encrypt data using the **RC4** algorithm. This encoding scheme needs an extra argument. The next argument after *data* is the encryption key. Encryption is symmetric. Re-application of the encoder command with the same key on the encryption result restores the original data.

- *-regsub*
  Encode special regular expression characters in a replacement string, such as *&* and *\1*, in such a way that they are not substituted by the next regular expression substitution operation and are afterwards again encoded in their original form.

- *-rfc2045*
  Encode in *base64* format, but enforce line breaks every 76 characters.

- *-rtf*
  Encode in such as way that the character sequence does not break the file syntax when inserted as string into an **RTF** file. Opening and closing curly brackets as well as backslash characters are escaped.

- *-savetodisk* (or *-save2disk*)
  Encode the data as header-complete **HTTP** response which triggers a receiving Web browser to pop up a save panel. If the optional extra argument is used, it is the default output file name pre-entered in the save panel.

- *-sql*
  Escape characters so that a well-formed **SQL** string results. The string does not contain the leading and trailing single quotes needed for use in an **SQL** statement.

- *-tcl*
  Encode a **PYTHON** interpreter value object, such as an integer, string, list or dictionary, into the standard string representation of the equivalent **TCL** object. The proper operation of this command depends on the **PYTHON** argument having the proper type. Since the command does not know the expected type, care must be taken when passing arguments - no automatic **TCL**-style conversion, for example from a string to a list or dictionary, is performed. These operations must be explicitly coded if needed. This encoding scheme is not supported in **TCL**.

- *-url*
  Encode for use in **URL**s, replacing space characters by + and other interfering characters by a *%xx* character triple, where the last two characters are hex digits for the ISO Latin1 character code. Additionally, linefeeds in the input in any format (Unix, PC, Mac) are standardized to a `CR/LF` sequence.

- *-urlbinary*
  The same as *-url*, except that linefeeds are not normalized. This is similar to the **JAVASCRIPT** function `encodeURI()`.

- *-urlbinaryparameter*
  An alias to *-urlbinarycomponent*

- *-urlbinarycomponent*
  The same as *-urlcomponent*, except that linefeed standardization of the encoded data is also suppressed.

- *-urlcomponent*
  The same as *-url*, except that characters in the set *&=?/;:* are also percent-encoded to avoid interference with the section structure of an **URL**. This is similar to the **JavaScript** **`encodeURIComponent()`** function.

- *-urlparameter*
  An alias to *-urlcomponent*.

- *-urlvbar*
  Encode for use in **URL**s, replacing space characters by + and other interfering characters by a *%xx* character triple, where the last two characters are hex digits for the ISO Latin1 character code. Additionally, line feeds in any of the standard forms (Unix, PC, Mac) are translated into a single vertical bar character.

- *-xml*
  Encode for usage in **XML** contexts. This encoding replaces the characters *>, <, ", '* and *&* by entities, but leave other characters, which would be converted into entities in **HTML** encoding, intact.

- *-zlib*
  Perform *zlib* compression on the data. The result cannot be directly stored as a *gzip*-compatible file. These files contain an extra header.

  The result of this command is likely to contain zero bytes and other special characters. Further processing of this data may require caution.

- *-zlib64*
  Perform *zlib* compression first, and then encode the result in the *-base64* format.

- *-zlib64url*
  Perform *zlib* compression first, then encode the result in the *-base64* format, and finally perform **URL** encoding of the result, which is required because *base64* can contain the characters +, = and /, which need to be escaped for **URL** contexts.

The *-zlib** encoding variants can also be invoked as the same command starting with *-zip**.

fcgi

```
fcgi subcommand ?args?
```

This command manages I/O when running an application script as an *FastCGI* (**FCGI**) Web application. Only selected interpreters provide this command, for example the standard **`tclcactvs`** and the stand-alone **`csweb`** variant.

There is currently no **Python** interface for this command.

In addition to providing the **`fcgi`** command, these interpreters also register additional pre-opened **Tcl** channels *fcgi-stdin*, *fcgi-stdout* and *fcgi-stderr*. Reading from or writing to these channels with the normal **Tcl** I/O commands directly moves the data to or from the **FCGI** server communication channels.

The following subcommands are supported:

```
fcgi accept ?redirect? ?autoexit?
```

This command enables **FCGI**-style socket communication with the Web server controlling this application and is usually the first command in an **FCGI** application.

If the *redirect* argument flag is set (it is on by default), the *stdout*, *stderr* and *stdin* **Tcl** channels are redirected to the corresponding **FCGI** multiplex socket. All normal **Tcl** I/O commands automatically use the multiplex socket instead of the standard channels if they are referring implicitly or explicitly to the standard channels. In addition, Web-typical properties like `E_GIF`, `E_VRML`, `E_PICT_IMAGE`, `E_EPS_IMAGE`, `E_SVG_IMAGE`, `E_EMF_IMAGE` and `X_GIF` are aware of this redirection and also send their output to the socket in case it is sent to one of the standard channels instead of a disk file. The current redirection status is mirrored in the read-only control variable **`::cactvs(fcgi_redirect)`**.

The standard output is sent to the remote browser. For normal Web applications, the output is typically **HTML** or some image format. Output must be started with a standard **HTTP** header (but no status code), which ends with an empty line. After that, the HTML or other content is added. Standard error gets sent to the Web server error log. This is simple text. Standard input contains form field data if the **CGI** call was from a form page. It can be decoded with the **`uncgi`** command. Query data specified as part of the **URL** (after a question mark) can be read from the environment variable **`::env(QUERY_STRING)`**.

If the *autoexit* flag is set (it is also set by default), the application automatically exits if the socket communication results in an error or **EOF**. The return value is a status code and zero in case a normal communication was initiated. In case automatic exiting is disabled, a negative status code is returned if there is an error.

The process can be terminated by the **FCGI** Web server module after some time of inactivity, or a fixed count of invocations. In addition, multiple processes with the same script may be started by the module in parallel. In case there is a load balancer, subsequent requests may even be sent to different physical hosts. For these reasons, it is generally not reliable to store global data in the script which should still be available in the next invocation. Instead, internal state like temporary objects should be cleaned up when a request has been executed and data which may be needed in a later processing step offloaded to storage using a lightweight session key or cookie as identifier - typically either to a disk file, or some distributed caching mechanism like **MEMCACHED** (see the **`memcached`** command).

Since **FCGI** scripts remain running after processing a request, edits to the script file are not immediately recognized. In order to pick up changes, all running interpreter processes with the script must be killed. They are automatically restarted by the **FCGI** Web server module on the next invocation.

```
fcgi finish ?redirect?
```

This command finishes the processing of a request. All buffered output is flushed to the server. The next request can then be waited for by a new call to **`fcgi accept`**.

If the *redirect* flag is set (it is set by default), any existing *stdout*, *stderr* and *stdin* redirections to the **FCGI** socket are canceled. All standard channels as seen by the script interpreter and certain image properties (see description of **`fcgi accept`** command) are reconnected to their original streams. The current redirection status is mirrored in the read-only control variable **`::cactvs(fcgi_redirect)`**.

```
fcgi puts ?-nonewline? ?channel? data
```

This command is the same as the standard **TCL** *puts* command, except that it operates on the *stdout* and *stderr* equivalents of the **FCGI** communication model. In case a channel is specified, only *stdout* and *stderr* are possible argument values. The command is primarily useful in case the automatic redirection feature of the **fcgi accept** command was not used. Output is only possible after the invocation of an **fcgi accept** command and before the issue of a closing **fcgi finish** command.

`fcgi read`

This command reads the **FCGI** equivalent of *stdin* until **EOF** is encountered and returns the data as a byte vector. In case **FCGI** redirection is active, the standard **TCL** *read* command can be used as an equivalent.

`fcgi redirect ?on/off?`

This command may be used to explicitly invoke or cancel the redirection of the standard **TCL** channels *stdin*, *stdout* and *stderr* to the **FCGI** equivalents. In case no boolean redirection argument is given, the current redirection status is reported. In case a new status is set, the command returns the previous status. The current redirection status is also mirrored in the read-only control variable **::cactvs(fcgi_redirect)**.

`fcgi status code`

This command can be used to set the exit status when the automatic exiting feature of the **fcgi accept** command is used. This command may be called any time.

Example:

```
while {[fcgi accept]>=0} {
   set data [read stdin]
   if {$data==””} { set data $env(QUERY_STRING) }
   uncgi $data params
   do_something $params(form_url_arg1) $params(form_url_arg2)
   fcgi finish
}
```

This code snippet shows a typical main loop in an **FCGI** application. Standard I/O redirection is used for convenience, and in case there are any communication errors with the Web server, or shutdown instructions, like closing of the *stdin* channel, are received from the server, the application exists. If the application scripts exits, it is (in a typical set-up, this depends on the Web server configuration) be restarted by the Web server at a later time when the next request comes in.

fetch

```
fetch ?-agent agent? ?-cookie cookie? ?-cookielist cookielist?
   ?-csrftoken token? ?-debug 0/1? ?-filename name? ?-header no/yes/exclusive?
   ?-modified time_in_secs? ?-password password? ?-referer referer?
   ?-setcookie cookie? ?-timeout nsecs? ?-tofile 0/1? ?-user username?
   ?-verification 0/1? ?-xheader1 header? ?-xheader2 header?
   ?-xheader3 header? url ?statusvar?
```

```
fetch(url=,?statusvariable=?,?agent=?,?cookie=?,?cookielist=?,?csrftoken=?
   ?filename=?,?header=?,?modified=?,?password=?,?referer=?,?setcookie=?,
   ?timeout=?,?tofile=?,?user=?,?verification=?,?xheader1=?,?xheader2=?,
   ?xheader3=?)
```

This command is used to retrieved data from network locations identified by **URL**s. The default action is to fetch the data and return is as uninterpreted byte data, but without protocol header information. The following options can be used to modify the command action:

- -agent *agent*
  Transmit a specific *User-Agent* **HTTP** header string. The default is an innocuous Mozilla browser name. If the parameter is set to an empty string, no agent name is transmitted.
- -cookie *cookie*
  When requesting the data, transmit the cookie string as part of the request in protocols which support this. Cookies may be required for access control, or encode modal information. This option cannot be combined with -*cookielist*.
- -cookielist *cookielist*
  A variant of the -*cookie* option. It cannot be combined with the -*cookie* option. The argument is a list of simple (no domain or expiration component, in the **`name=value`** format) cookie specifications, which are sent as a bundled collection.
- -csrftoken *token*
  This option simplifies the handling of **CSRF** tokens. A token specified here is added to the **HTTP** header in a *X-CSRF-Token:* header line.
- -debug 0/1
  If set, print debug information about the connection and data transfer on standard error.
- -filename *filename*
  The retrieved data is written to the specified file, and not to a *temp* file which is the default if no filename is set and output is routed to a file. This flag implies -*tofile 1*.
- -header *no/yes/exclusive*
  This option determines whether header information should be retrieved. By default, header data is discarded. The *exclusive* mode only reports the header and discards any body information. This option only makes sense for protocols which differentiate between header and body information.
- -modified *time_in_secs*
  Only fetch the data if it has been modified since the specified time stamp. The format of the time value is seconds since epoch, this is the same type as returned by the standard **TCL** **`clock seconds`** format.
- -password *pwd*
  A password to be used with the user name to supply credentials to the remote server. Alternatively, it can also be specified as part of the **URL** in standard notation.
- -referer *referer*
  A referrer string to pass to the server. Yes, this is written with a single R.
- -setcookie *cookie*
  When requesting the data, include header lines to set the specified cookie in protocols which support this.

- -timeout *nsecs*
  Set the timeout in seconds. The default timeout is 30 seconds. If the data could not be retrieved completely within the specified time frame, an error is produced.
- -tofile *0/1*
  Output to a temporary file instead of an in-memory image. The return value is the handle of a **Tcl** or **Python** file channel. Option *-file* is an alias.
- -user *username*
  A user name to present to the remote server. Alternatively, it can also be specified as part of the URL in standard notation.
- -verification *0/1*
  This flag is set by default and forces a certificate verification when using encrypted connections such as *https*. If not set, verification is skipped. This is faster, and may be necessary for sites which have ill-configured certificates, but of course may hide site identity problems.
- -xheader[123] *headerline*
  These parameters allow the injection of custom lines into the **HTTP** header. The line data should be provided without linefeeds, and must contain the header field name.

The *url* argument is a standard Internet **URL**. All standard protocol drivers also support the transport of user id and password information. Internally, this command uses the **CURL** library.

If a status variable parameter is specified, it is interpreted as the name of a **Tcl** array variable or **Python** dictionary which is created if necessary, reset and filled with status information. Its elements are *size* (the download data size, without header), *lastmodified* (the remote file modification date), *location* (the last download **URL**, which can be different from the original retrieval **URL** in case of redirects), *mimetype* (the **Mime** type of the data, as perceived by the remote server), various cookie data variants (see below) and finally *status* for the integer return status code.

The elements for the cookie data element encoding variants are named *cookies* (a list/tuple of all received cookies in Netscape format), *cookiedicts* (the same as list/tuple of dictionaries, with dictionary keys identifying the cookie field data components), and individual per-cookie elements *cookie%d* and *cookiedict%d,* where the placeholder is the cookie index starting with zero. The Netscape cookie format is a tab-separated string of six elements encoding the cookie domain, its domain access flag, the path, secure access flag, expiration date, cookie name and value. Splitting these strings to access individual data items should be done explicitly on the tab character since implicit list conversion is not reliable for this encoding. In the dictionary list and individual cookie dictionary encoding variants, the dictionaries contain the same information properly isolated under the keys *domain*, *domainaccess*, *path*, *secureconnection*, *expiration*, *name* and *value*. The individual cookie strings are simplified data representations pre-formatted in the style `"name=value"` or "`name=value; Path=path"` (the latter only if an explicitly path was specified) which is usually the proper form to use when constructing a custom **MIME** header with a *Cookie:* or *SetCookie:* tag.

If the program is run in *-header exclusive* mode, the command result is a list of the status variable values *size*, *lastmodified*, *mimetype*, *code*, *location* and *cookies*, in the order listed, instead of the fetched data.

The modification time as well as cookie expiration dates in the status are returned as an integer (seconds since midnight, Jan 1st, 1970, *epoch*), suitable for formatting by means of the standard **TCL clock format** command. If a modification or expiration date could not be determined, the reported time value is minus one.

The mime type information may contain, after a semicolon separator, additional encoding information if reported in this fashion by the server.

When the command is executed, the thread-local **::cactvs(lookup_timeout_occurred)** flag is reset. If the command fails because of a server timeout, but no other causes, the flag is set.

Example:

```
set data [fetch -timeout 10 ftp://$user:$passwd@$host/$path/$file v_status]
set mimetype $v_status(mimetype)
```

filecheck

```
filecheck type filename
filecheck(type=,filename=)
```

This command is used to check formats and attributes of files. The result is either boolean 1, if the file is of the checked type, or 0 if it is not. In case of errors, such as a non-existing file, a standard error condition is raised. These file type checks are currently supported:

- *binary*
  Check whether file looks like a binary (not text) file.
- *bzip2* (or *bzip*)
  Check whether file is *bzip2*-compressed.
- *compressed*
  Check whether file is of a recognized compression type.
- *directory*
  Check whether path is an existing directory.
- *dll*
  Check whether file is a shared library of any type (shared, **DLL**, bundle). Same as *sharedlib* test.
- *filter*
  Check whether file is a **CACTVS** filter definition file.
- *factory*
  Check whether file is a **CACTVS** factory definition file. Factory parameter files are not factory definition files.
- *file*
  Check whether path is a normal file (and not a directory, link, socket, etc.)
- *gif*
  Check whether file is a **GIF** image.

- *gzip*
  Check whether file is *gzip*-compressed.
- *html*
  Check whether file is a **HTML** file.
- *image*
  Check whether file is an image in a suitable Web format (**GIF**,**JPEG**,**PNG**)
- *jpeg* (or *jpg*)
  Check whether file is a **JPEG**-encoded image.
- *link*
  Check whether path is a link (and not a file, directory, socket, etc.)
- *local*
  Check whether file resides on a local file system. Not supported on all platforms.
- *object*
  Check whether file is a compiled object file.
- *pdf*
  Check whether file is a **PDF** file.
- *postscript*
  Check whether file is a **PostScript** file.
- *png*
  Check whether file is a **PNG**-encoded image.
- *property*
  Check whether file is a **Cactvs** property definition file. Both the old keyword/value format and the new **XML** format are recognized.
- *repository*
  Check whether file s a **Cactvs** repository in **SQLITE** format.
- *sharedlib*
  Check whether file is a shared library of any type (shared, **DLL**, bundle). Same as *dll* test.
- *shellscript*
  Check whether file is a shell script.
- *special*
  Check whether file name is one of the magic names recognized by **Cactvs**, such as *stdin* or *stdout*.
- *sqlite*
  Check whether file a s **SQLite** version 3 database file.
- *station*
  Check whether file is a **Cactvs** station definition file. Station parameter files are not station definition files.

- *tmpfile*
  Check whether file is residing in the current *tmp* directory.
- *ucs2encoded*
  Check whether file is encoded in 16-bit Unicode, in either of the two possible byte orders.

These tests operate by examining the first bytes of a file, not by a simple match of suffixes. File checks which rely on platform-specific binary data layout information only work for the current platform. On a Sun, a OSX bundle will not be recognized, and vice versa.

insidepolygon

```
insidepolygon {x y} {{x1 y1} {x2 y2} {x3 y3}...}
insidepolygon x y {{x1 y1} {x2 y2} {x3 y3}...}
insidepolygon x y x1 y x2 y2 x3 y3...
insidepolygon((x,y),((x1,y1),(x2,y2),(x3,y3),...))
insidepolygon(x,y,((x1,y1),(x2,y2),(x3,y3),...))
insidepolygon(x,y,x1,y1,x2,y2,x3,y3,...)
```

A boolean test to determine whether the test point is on the inside of the polygon defined by the points **`{x1 y1}`**... The first and last points of the polygon are assumed to be connected, so one should not supply a start and end point duplicate. The minimum polygon size is three points, and there is no upper limit. The point encoding is determined by the number of arguments (2,3, or more).

ldelete

```
ldelete ?mode? list pattern ?pattern?...
```

Delete elements from a list. The first optional parameter selects the match mode for the patterns. If can be one of -*exact*, -*nocase*, -*substring*, -*regexp*, -*glob* or -*index*. The standard option list terminator “--” may also be used. The default mode is *exact*. The index mode expects element indices starting with 0 as pattern arguments.

All list elements which match any of the patterns are removed. The command returns the shortened list, preserving the order of the surviving elements.

This command is not directly supported in the **Python** interface - the function can easily be implemented with standard list comprehensions (see for example *https://stackoverflow.com/questions/20349125/removing-a-pattern-from-a-list-items-in-python*)

Example:

```
ldelete -regexp [list a aa ad] {a[abc]+}
```

returns “a ad”.

lineintersect

```
lineintersect x1 y1 x2 y2 x3 y3 x4 y4
lineintersect {x1 y1} {x2 y2} {x3 y3} {x4 y4}
lineintersect(x1,y1,x2,y2,x3,y3,x4,y4)
lineintersect((x1,y1),(x2,y2),(x3,y3),(x4,y4))
```

Test whether the line segments from x1/y1 to x2/y2 crosses the line segment from x3/y3 to x4/y4.

The return value is a simple boolean value. It does not report the intersection coordinates.

lsearch

```
lsearch ?modeflags? list pattern
```

This is an extended, but compatible version of the standard **Tcl** **`lsearch`** command.

The mode flags may be any combination of -*exact*, -*nocase*, -*substring*, -*regexp*, -*glob*, -*first*, -*all* and the option terminator "--". The default mode is the combination of *glob* and *first*, the same as for the standard **`lsearch`** command.

Just as the standard command, matching list indices are returned. In mode *first*, which corresponds to the default behavior, the result is the list index >= 0 of the first matching element, or -1 if the pattern cannot be found. In mode *all*, a list of the indices of all matching list elements is returned, or an empty list if no element matches.

The command is not directly supported in the **Python** interface - use the standard list comprehension and pattern match functions for this functionality.

Example:

```
lsearch -exact -all {a b c a b c} a
```

returns "0 3".

lsum

```
lsum nmbers...
lsum(?number,...)
```

Sum up all number arguments and return the result. All arguments must be numbers.

`sum` is an command alias.

lvardelete

```
lvardelete ?mode? listvar pattern ?pattern?...
```

Delete elements from a list variable. The first optional parameter selects the match mode for the patterns. If can be one of -*exact*, -*nocase*, -*substring*, -*regexp*, -*glob* or -*index*. The standard option list terminator "--" may also be used. The default mode is *exact*. The index mode expects element indices starting with 0 as pattern arguments.

All list elements which match any of the patterns are removed from the variable. Additionally, the command returns the shortened list, preserving the order of the surviving elements.

The command is not directly supported in the **Python** interface - use the standard list comprehension and pattern match functions for for this functionality.

Example:

```
set mylvar [list a b c d]; lvardelete myar b d
```

returns "a c" and sets the variable to the same value.

mail

```
mail ?-attachment filename? ?-debug 0/1? ?-file filename?
   ?-from address? ?-secure 0/1?
   ?-smtphost hostname? ?-smptpassword password? ?-stmpport port?
   ?-smptuser userid? addresslist ?subject? ?message?
```

```
mail(addresses=,?subject=?,?message=?,?attachments=?,?debug=?,?filename=?,
   ?from=?,?secure=?,?smtphost=?,?smtppassword=?,?smtpport=?,?smtpuser=?)
```

Send a simple iso-encoded plain-text email message to one or more recipients. The only required argument is the recipient email address list. Without subject and body arguments, an empty message is sent. If the recipient address list is an empty list, or only contains empty elements, the command silently does nothing. Address list elements which are not empty strings and ignored must pass a simple email address pattern test before a mail delivery attempt is made.

The optional *smtp** arguments specify the access parameters to the mail host. If these options are not set, they are read from the corresponding control variable elements, i.e. **`::cactvs(smpt_host)`**, **`::cactvs(smpt_port)`**, etc. If the *-secure* option is set, an attempt is made to use encrypted **SMPTS** communication instead of the plain **SMTP** protocol, though the initial protocol negotiation still uses plain **SMTP** if the **SMTP** port is not explicitly set to 465. In absence of an *-from* argument, the sender address is copied from **`::cactvs(user_email)`**. There is no required relationship between the *from* address and the mail host access parameters. The default **SMTP** port used by this command is 587, not the old standard 25.

Experience teaches that talking to mail servers can be tricky. When the *-debug* option is set, trace output from the communication attempts is printed on standard error. This is highly useful to pinpoint connection problems.

Finally, the *-file* option allows the direct upload of an existing, readable file as the message. The file contents are sent as message data, not as an attachment. If both an upload file and a body argument are specified, the file is inserted first.

The message content is sent as plain Latin1 (ISO8859-1) encoded message. The nature of line-break characters in the message is not preserved, and additional line breaks are inserted if any line is longer than 998 characters, the maximum line length in a standard **SMTP** message.

This command currently neither supports *pop-before-smtp* authorization, nor the sending of attachments, though they are accepted as arguments. In the **TCL** command version, multiple attachments are specified by repeating the -attachment command. In **PYTHON**, an attachment file name tuple is expected, because named function arguments cannot be repeated.

The argument order of the **PYTHON** command is intentionally different to allow sending of messages without setting more exotic options. All arguments after the first three must be specified as named arguments.

Example:

```
mail -smtphost smtp.gmail.com -smtpuser $user -smtppassword $pw \
   -secure 1 wdi@xemistry.com "Hi" "This\nis a message\n"
```

Above sample command sends a message via Google mail. Of course, a Gmail account bound to the user and password variables must exist for this command to succeed.

mailcap

```
mailcap mimetype
```

```
mailcap(mimetype)
```

Return the standard opener/viewer for files of the specified **MIME** type. The result is dependent of the local configuration.On Windows, the command attempts to find a suitable opener in the registry, and if it is found, the return value usually contains Windows-specific placeholder tags.

Example:

```
mailcap chemical/pdb
```

might return something like “rasmol -pdb %s”

parse

```
parse base64 arg
parse binary arg
parse casrn arg
parse color arg
parse datauri arg
parse date arg
parse dictionary arg
parse doi arg
parse domain arg
parse element arg
parse elementlist arg
parse email arg
parse formula arg
parse hash arg
parse hex arg
parse inchi arg
parse inchikey arg
parse ipaddr arg
parse jme arg
parse json arg
parse lillyrule arg
parse linenotation arg
parse list arg
parse minimol arg
pasre nativequery arg
parse orcid arg
parse propertyname arg
parse propertylist arg
parse query arg
parse queryformula arg
parse retrievallist arg
parse sln arg
parse smarts arg
parse smiles arg
parse superatom arg
parse unii arg
parse uuid arg
parse xml arg
parse(class=,data=)
```

This command checks whether the argument can be parsed as the data type indicated. The result is a boolean value. No error is raised if the parsing fails, and the parsed data structure is discarded. The following data types are currently understood:

- *base64*
  base64-encoded data
- *binary*
  The file is binary, i.e. it contains bytes outside the Iso-Latin1 character set which are not printable, whitespace or line control characters.
- *casrn*
  a Chemical Abstracts registry number. Only the syntactical correctness, including the check digit, is verified, not whether the number is valid.
- *color*
  a valid color specification, either as a recognized name, or in hash notation.
- *datauri*
  a valid data **URI**.
- *dataurl*
  this is an alias to subcommand *daturi*.
- *date*
  a date, time or datetime value of which the format can be automatically detected and parsed
- *dictionary*
  a **Tcl**-style dictionary, i. e. either a **Tcl** dictionary object, or a properly formed list of keyword and value pairs, as they are also used in some **Cactvs** interface functions, for example the *parameters* attribute of property definitions.
- *doi*
  a properly formed **DOI** (digital object identifier).
- *domain*
  a properly formed domain name. Only the generic syntax is checked, no **DNS** look-up is performed. The domain must contain at least one dot.
- *element*
  an element symbol. Case is disregarded.
- *elementlist*
  a list of element symbols, as they are for example used in the list field of the `A_QUERY` property to specify an element list for substructure matching. Superatoms and other element extensions are not allowed here.
- *email*
  a properly formed email address.

- *formula*
  a plain molecular formula, without query expressions such as count ranges or pseudo elements.
- *hash*
  a CACTVS 64bit hash code
- *hex*
  a hexadecimal number. Extra white space is allowed.
- *inchi*
  an INCHI string. This is only supported in interpreter versions with INCHI support. INCHI support needs to be compiled-in. It cannot be loaded as I/O module.
- *inchikey*
  a INCHI key/hash string. This is only supported in interpreter versions with INCHI support. INCHI support needs to be compiled-in. It cannot be loaded as I/O module.
- *ipaddr*
  a properly formed IP4 address in dot-separated bytes in decimal encoding.
- *jme*
  a MOLINSPIRATION Java Molecule Editor structure string. This requires the presence of the JME I/O module. An attempt is made to auto-load it if it is not already in memory. If the module cannot be loaded, an error results, even if the string is syntactically correct.
- *json*
  a properly formatted JSON blob
- *lillyrule*
  Check syntax of a match expression in Bruns/Watson notation (J. Med. Chem. 2012, 55, 9763-9772
- *linenotation*
  a string which can be decoded as standard structure line notation, as recognized by `ens create` without any extra flags.
- *list*
  a properly formed TCL-style list
- *minimol*
  a CACTVS Minimol in binary or hex-encoded variants
- *multilineascii*
  A multi-line ASCII or Latin1 string. It must contain line breaks, and cannot contain non-printing characters other than standard whitespace.
- *nativequery*
  a query expression, as used in `molfile scan` and other scan commands, in native toolkit notation. This excludes queries in the Lilly format.
- *orcid*
  a properly formatted ORCID (www.orcid.org) identifier. The general format and the check digit are verified, but not whether this ID has been issued.

- *propertyname*
  a string which conforms to the standard syntax of **CACTVS** property names. It is not checked whether the property is defined.
- *propertylist*
  a list of toolkit property names. An attempt is made to auto-load properties for which no definition is in memory. This command can also be spelled as *proplist*.
- *query*
  a query expression, as used in `molfile scan` and other scan commands. Both the native toolkit style and the Bruns/Watson Lilly notation are recognized.
- *queryformula*
  a formula query expression, as used in the `molfile scan` and other scan commands.
- *retrievallist*
  a data retrieval list, as used in `molfile scan` and other scan commands.
- *sln*
  a valid Sybyl line notation string. This requires the presence of the **SLN** I/O module. An attempt is made to auto-load it if it is not in memory. If the module cannot be loaded, an error results, even if the string is syntactically correct.
- *smarts*
  a valid **SMARTS** string, including Recursive **SMARTS**
- *smiles*
  a valid **SMILES** string
- *superatom*
  a superatom which can be decoded from the current superatom table entries without ambiguity
- *uuid*
  an **UUID** in standard 8-4-4-4-12 notation, syntactically checked in case-independent fashion
- *unii*
  a **FDA UNII** registry number in all-lower or all-upper case. Only the syntactical correctness, including the check digit or letter, is verified, not whether the **UNII** is valid.
- *xml*
  a well-formed **XML** document

passwd

```
passwd encode cleartext_pw ?salt?
passwd decode encrypted_pw cleartext_pw
passwd(mode="encode",password=,?salt=?)
passwd(mode="decode",password=,cleartext=)
```

The first command variant encodes a clear-text password with the standard Unix `crypt()` algorithm. A salt value for the password generation may be specified as exactly two letters from the set “a-zA-Z0-9./”. If no salt is specified, a random value is used. The command returns the encoded version of the clear text.

The second variant returns 0 or 1, depending on whether the clear text password matches the encrypted version or not.

This command is supported on Windows, the **`crypt()`** function is provided by a compatibility function compiled into the library code.

Example:

```
passwd encode topsecret XX
```

yields something like “XX9Kadd0cpq.o”.

```
passwd decode XX9 topsecret
```

returns 0, while

```
passwd decode XX9Kadd0cpq.o topsecret
```

returns 1. Note that without the specification of a salt parameter the encoded result is different each time in a random fashion. However, the password check works without knowing the salt with encrypted passwords encoded with each possible salt.

post

```
post ?-accept accepts? ?-agent agent? ?-boundary text? ?-cookie cookietext?
   ?-contenttype mimetype? ?-cookie cookietext? ?-csrftoken token?
   ?-debug 0/1? ?-fieldcontenttype type? ?-host hostname?
   ?-language languagecode? ?-password pwd?
   ?-raw? ?-referer referer? ?-setcookie cookie? ?-timeout secs?
   ?-uploads uploaddictionary? ?-user username? ?-verifyhost 0/1
   ?-xheader1 headerline? ?-xheader2 headerline? ?-xheader3 headerline?
   url ?fielddictionary/rawdata? ?statusvariable?
```

```
post(url=,?data=?,?accept=?,?agent=?,?boundary=?,?cookie=?,?contenttype=?,
   ?csrftoken=?,?debug=?,?fieldcontenttype=?,?host=?,?languate=?,
   ?password=?,?raw=?,?referer=?,?setcookie=?,?timeout=?,?uploads=?,?user=?,
   ?verifyhost=?,?xheader1=?,?xheader2=?,?xheader3=?,?statusvariable=?)
```

Assemble the code for a **HTTP** or **HTTPS POST** message in (by default) *multipart/form-data* encoding, transmit it to a remote server and read the response.

In the standard command mode, the field dictionary parameter is a list of field names and their contents to include in the **POST** message. The transfer format is binary, so it is possible to send byte array data. The default individual field mime format is *text/plain*, but **PNG** and **GIF** images as field data are automatically recognized and sent as *image/png* and *image/gif,* respectively. In order to mirror the data encoding characteristics of file upload **HTML** form elements faithfully, an optional uploads dictionary can be provided. It is a key/value collection where the keys are field names that are file upload elements in the emulated Web form. They also need to be contained in the field dictionary as field name / field data pair. Field names only present in the uploads directory are ignored. The upload dictionary values are the *filename* attribute of the transmitted fields. The encoding of transmitted data for fields in this dictionary is modified so that the content type is always set to *application/octet-stream* instead of *text/plain*. Whether the configuration of the upload dictionary is required or not when emulating the submission of forms with upload fields depends on the extent of the data analysis performed on the side of the contacted server, but its use does not potentially introduce new problems. File contents are not automatically opened and read. Their contents must be provided in the field dictionary.

If the *-raw* flag is set, no interpretation of the field dictionary parameter takes place. Instead, the value of this parameter is transmitted verbatim as byte sequence in the message body. In this mode, it is assumed that it already contains all field formatting and only needs to be augmented with the **HTTP** header data. The **Tcl** version of the command uses this flag as a stand-alone parameter, while in the **Python** version it requires a boolean value specification.

The other optional parameters allow the setting of additional fields in the header of the **POST** command. The `-boundary` attribute is the **MIME** separator string. Its default value is a random string that is highly unlikely to occur in any transmitted data. The default content type for the form submission is *multipart/form-data*, but this may be adjusted with the *-contenttype* parameter. The short argument forms *multipart* (or *formdata*) and *urlencoded* select the standard encodings *multipart/form-data* and *application/x-www-form-urlencoded*. Other values of this option are sent verbatim to the server after a *Content-Type:* keyword. The additional attributes of file upload fields are ignored in *urlencoded* mode.

If the *-accept* option is set, this defines the accepted encoding formats of the data sent from the server. The value corresponds to the the **HTTP** *Accept:* header format. The argument of the *-language* option is passed to the *Accept-Language:* **HTTP** header field. By default the *Host:* **HTTP** header is extracted from the destination **URL**, but a custom value can be set with the *-host* option. The *-xheader[123]* parameters allow the injection of custom lines into the **HTTP** header. The line data should be provided without linefeeds, and must contain the header field name. The *-csrftoken* option simplifies the handling of **CSRF** tokens. A token specified here is added to the **HTTP** header in a *X-CSRF-Token:* header line.

Field content is normally posted with a "*text/plain; charset=utf8*" type header, except for data which is recognized to be in another well-known format (like **GIF** or **PNG** images), where the proper standard type is substituted. The `-fieldcontenttype` attribute can be used to override this default typing. If it is set to an empty string (or Python `None`), the fields are posted without type information. Individual typing of fields is currently not supported.

If the *-verifyhost* option is set to `false`, and the connection uses SSL encryption, the remote host identity is not rigorously checked against its SSL certificate. This allows interaction with hosts which used mis-configured SSL certificates, but this lenient approach is obviously a potential security issue and should only be used when necessary. The other parameters have the same meaning as in the related `fetch` command.

If the name of a status variable is specified, it is created or reset in the local namespace, and filled with the elements *status* (the **HTTP** server error code, 200 for normal retrieval), *size* (the length of the received content in bytes), *cookies* (a list of all cookies received in Netscape string format), *cookiedicts* (a list of cookies in dictionary form), *location* (the response **URL**, which can be different from the command argument in case of server redirection), *lastmodified* (the modification time stamp of the result data, if available, -1 otherwise) and *contenttype* and *mimetype* (the **MIME** type of the response). In addition, every cookie is individually stored as element *cookiedict%d*, starting with index zero and with the seven elements *domain*, *domainaccess*, *path*, *secureconnection*, *expiration*, *name* and *value* that are contained in the standard *Netscape* cookie encoding. The variable element names are compatible to those used in the `fetch` command.

It is possible to omit the field dictionary argument if an empty message is to be sent. The **URL** may also contain field arguments in **URL** encoding, but whether these are recognized in addition to the data in the field dictionary depends on the destination server configuration.

When the command is executed, the thread-local `::cactvs(lookup_timeout_occurred)` flag is reset. If the command fails because of a server timeout, but no other error causes, the flag is set.

The argument order is intentionally different for **Python** in order to allow simple submissions without named arguments. All arguments after the *data* item must be set as named arguments.

Example:

```
set data [post -contenttype urlencoded -cookie $c -debug 0 \
-csrftoken [dict get $dcsrf value] \
-referer https://mcule.com/search/ \
https://mcule.com/search/post \
[dict create csrfmiddlewaretoken [dict get $dcsrf value] mode exact structure_2
$mdata]]
```

prod

```
prod args...
prod(?arg?,...)
```

Interpret the arguments as numeric values and compute their product. If called without arguments, the function returns one.

python

```
python eval expression
python exec statements
```

Execute a **Python** command or expression from **Tcl**. This command is only available in interpreters which include a **Python** subsystem. This command is not supported in **Python**, but on that side, comparable `tcl()` and `tcl1()` method exist.

For the *eval* variant, only a single statement may be processed, and the result of that statement is returned to **Tcl**. Longer statement blocks can be processed with the *exec* command variant, but in that case the result is always an empty string if no error occurs. This is a limitation of the **Python API**. In case an error is raised, its description is transferred back as **Tcl** result string, and the command also reports an error.

The **Python** interpreter version is currently 3.6. It is fully featured with access to the complete **Python** standard library, and can be instructed to import additional modules.

At this time, only a single primary **Python** interpreter exists in **Python**-enabled toolkit applications. It is mutex-protected and can be called from multiple **Tcl** threads, but not simultaneously. Property computation **Python** sub-interpreters are internally used, but not accessible via this command.

The **Python** interpreter in the standard toolkit context (but not when loaded as **Python** module) has already imported the `pycactvs` and `sys` **Python** modules. All `pycactvs` functions and other objects are explicitly imported and can by used without a module prefix. The `pycactvs.tcl()` and `pycactvs.tcl1()` functions allow scripts to double back and execute a `Tcl` command in the main `Tcl` interpreter from within **Python**.

Examples:

```
set five [python eval "2+3"]
```

```
python exec "print('hello world')"
set eh [python eval "tcl('ens create CCC')"]
```

The first example computes a simple expression in **Python** and returns the result. It is automatically efficiently encoded as a **Tcl** integer object.

The last example executes a command in the **Python** interpreter, which itself turns back to the **Tcl** interpreter to execute an **`ens create`** statement. Toolkit objects are shared between the **Python** and **Tcl** interpreters. The newly created ensemble object is known and accessible in both languages simultaneously.

quote

```
quote arg
```

Perform an efficient string quoting operation of a Latin1 string. The result is a string which contains only 7-bit printable characters. Characters outside this range are encoded as backslash-encoded octets or standard backslash escape sequences such as '\t' or '\\'. The code for this command was copied from the **Tcl** Netscape plug-in.

The command is not supported in the **Python** interface because of the different string handling semantics in that language.

random

```
random limitval ?sequencesize ?seedval|reset?
random seed ?seedval?
random reset
random hex32 ?seedval/reset?
random hex64 ?seedval/reset?
random hex128 ?seedval/reset?
random uuid
random(limitval,?sequencesize?,?seedval|"reset"?)
random("seed",?seedval?)
random("reset")
random("hex32",?seedval|"reset"?)
random("hex642",?seedval|"reset"?)
random("hex128",?seedval|"reset"?)
random("uuid")
```

Generate integer pseudo-random numbers. In the simplest command variant, the *limitval* parameter is an upper exclusive bound. The returned pseudo-random number is in the range 0 to *limit-1*. By default, only a single number is returned. If a non-empty *sequencesize* parameter is added, the specified number of pseudo-random numbers is returned as a list. If the third optional parameter is the string *reset*, the random number generator is reset to its default start state. Alternatively, a numerical seed value can is used to initialize it to a different initial state before computing the random numbers..The random numbers are always the same sequence for a given initial state. If this is not desired, some pseudo-random seed, such as the current clock click value, should be used.

The **`random seed`** and **`random reset`** command variants can be used to manipulate the state without starting the random number generation. The **`seed`** subcommand without a seed argument uses a combination of a high-resolution timer and the process ID as seed value and is the recommended method to set up the generator for non-repeatable randomness.

The **`random hexxx`** variants directly return hex string encodings of 32, 64 or 128 bit integers, which possess fixed string lengths of 8, 16 and 32 characters, respectively. Optionally, this operation again can be combined with a reset operation or a seed value.

The **`uuid`** subcommand generates a new **UUID**. This function is independent of the normal random state.

This command is thread-aware. Every script thread has its own random state, and random number generation or state manipulation in one thread has no effect on any other interpreter thread. The **`uuid`** subcommand is the sole exception - it is not thread-aware, and not coupled to the thread-local random state of the other commands.

This random number command overrides the standard **TclX** command of the same name and provides a superset of its functionality, in addition to thread-awareness.

Example:

```
random seed
foreach r [random 16 8] {...}
```

rpc

```
rpc bynumber servicenumber
rpc byname servicename
rpc(mode='bynumber',arg=)
rpc(mode='byname',arg=)
```

Look up **RPC** services by name or service number. If the service can be found, the result is a list consisting of the service name, the service number, and a list of all name aliases.

This command is supported on Linux/Unix only, and only in special builds.

screen

```
screen ?-aromatch lenient|normal|strict? ?-frags?
   ?-hydrogens ignore|match|mark? ?-mode ligands|smarts? ?-nobondorder?
   ?-tauto? ehandle patternlist ?exactmatchvar?
```

```
screen(ens=,patterns=,?aromatch=?,?frags=?,?hydrogens=?,?matchvariable=?,
   ?mode=?,?nobondorder=?,?tauto=?)
```

Perform fragment-based bit screening on an ensemble object. If the *-frags* option is not set, he result is a bit vector of the same length as the pattern list, with bits set to zero or one depending on whether a pattern fragment matches or not. If *-frags* option is set, the return value is a list of the patterns which match, copied verbatim from the pattern list argument.

If the optional exact match variable parameter is given, it is the name of a **Tcl** or **Python** variable which is set to one if the input ensemble is of exactly the same connectivity (without stereochemistry or isotopes) as one of the pattern fragments. If no pattern is an exact match, it is set to zero.

The expected format of the pattern list depends on the match mode. The default match mode is *ligands*, which can be explicitly set by supplying the argument to *-mode*. The other possible match mode is *smarts*, set by *-mode smarts*.

In *smarts* mode, the patterns are normal **SMARTS** strings. These are cached internally, so if patterns are reused, and the total number of patterns is not too large for the cache, they are not decoded anew for every command invocation.

In *ligands* mode, a simpler pattern language which is faster to decode and match is used. The patterns are either two element symbols, connected by a bond symbol (- for single bond, = for double, * for triple, # for quadruple, ~ for aromatic - note that this is different from **SMILES**!), or an element, followed by a sequence of bracketed ligands, each with a preceding bond symbol. Patterns like these describe one central atom (the first element in the pattern), with one or more ligands. Examples are *"P=O"* or *"C(-Cl)(=O)"* or *"C(-H)(~C)(~C)"*. For internal reasons, all multiple bonds in a pattern must be written before any aromatic bond.

The -*aro* option controls how aromatic systems in the ensemble are matched. The default is *normal*, meaning that single and double bonds in **SMARTS** patterns match aromatic bonds in the ensemble, regardless of their Kekulé bond order, and single bonds in *ligands* patterns match aromatic ensemble bonds. In *strict* mode, non-aromatic bonds in the pattern do not match aromatic ensemble bonds. This mode has an effect only for *smarts* matching, not *ligand* matching. The aro match mode *lenient* only has an effect on matching *ligand* patterns. If not selected, ligand pattern double bonds do not match aromatic ensemble bonds. If the mode is *lenient*, they do.

If the -*tauto* flag is set, the matching of both ligand and **SMARTS** patterns is further modified. In *ligands* mode, tautomeric ensemble bonds (as per property `B_ISTAUTOMERIC`) are excluded from any match which requires an exact bond order match. In *smarts* mode, the matched fragment must be compatible with a tautomeric form with shifted bond orders and relocated hydrogen atoms. This flag is intended to be used to construct a screening bit string for tautomer-tolerant substructure searches. Because less patterns can be positively identified to be matching under these circumstances, these bit strings have generally less bits set, and are therefore less effective filters.

If the -*nobondorder* flag is set, bond orders of pattern fragments of both types are ignored.

The -*hydrogen* option controls how patterns with hydrogen atoms are handled. By default, in mode *match*, they are matched like any other pattern. In mode *ignore*, patterns with hydrogen never match. This is the mode to use if a screen vector for superstructure search is constructed. In mode *mark*, any pattern which contains a hydrogen atom reports a match, and any pattern which does not a mismatch. This mode is intended to be used for construction of bit masks for the removal of any bits from a screen vector which are hydrogen-dependent. This allows for example the re-use of a substructure screen vector for superstructure search.

The **Python** version of the command intentionally has a different argument order than the **Tcl** version. Additionally, single-word arguments in the **Tcl** version must be specified as named arguments with a boolean value. All parameters after the pattern set must be passed as named arguments.

sqsum

```
sqsum args...
sqsum(?arg?,...)
```

Interpret the arguments as numeric values and compute the sum of the squared values.

stddev

```
stddev arg arg ?args?...
stddev(arg,arg,?arg?,...)
```

Interpret the arguments as numeric values and compute their standard deviation. A minimum of two arguments is required.

sum

An alias for **lsum**.

tmpdir

```
tmpdir
tmpdir()
```

This command returns the current potentially application-specific directory for temporary files (*/tmp* or *C:/temp* or similar).

tmpname

```
tmpname ?prefix? ?directory? ?suffix?
tmpname(?prefix=?,?directory=?,?suffix=?)
```

Generate a valid name for a temporary file. The file name prefix, the suffix as well as the directory may be specified. If these parameters are omitted, or set to empty strings (including **None** for **Python**) a temporary file in the standard *tmp* directory and with a file name prefix derived from the application name (obtained via invoking the command **infox appname** in the current **Tcl** interpreter, where the standard **Tcl** features to configure its results apply) is generated.

Example:

```
tmpname myapp {} .res
```

returns something like */tmp/mya2462.res*, dependent on the operating system and local set-up.

uncgi

```
uncgi ?-noreset? cgi_string ?variable? ?untrimmedfields?
   ?fieldname default?...
uncgi(cgi_string=,?defaults=?,?noreset=?,?untrimmedfields=?,?variable=?)
```

This command decodes **CGI** data, as it is delivered to a **CGI** application from data input in a WWW form via the Web server. It automatically recognizes the two standard transfer formats *application/x-www-form-urlencoded* and *multipart/form-data* and adjusts the decoding process accordingly.

The *cgi_string* input data can be obtained either from the **QUERY_STRING** environment variable (**PUT** forms, or direct **CGI URL**s), or from reading the *stdin* channel (**POST** forms). A typical code snippet in case the data submission method is not known is:

```
fconfigure stdin -translation binary
set data [read stdin]
if {$data==""} { set data $env(QUERY_STRING) }
```

```
uncgi $data params
```

The return value of this command is a dictionary with name/value pairs. This dictionary can be read into an array variable by the standard **`array set`** **TCL** command. If the optional *variable* parameter is used, an array variable of that name is automatically created or reset and filled with the decoded data in the form of array elements.

By default, leading and trailing white space is removed from the decoded data. Fields for which this white space removal should not be performed can be listed as a standard **TCL** list in the *untrimmedfields* parameter. These fields are decoded without any processing.

Note that it is generally advisable to set the *stdin* input channel to binary mode before reading the **CGI** data. On Unix, this is likely to be a no-op, but the transfer of binary data on Windows usually does not work at all without this setting.

For security reasons, certain **XML**/**HTML**-style tags, including their inner text if they possess it, are automatically removed from the input data. The currently deleted tag list is **`<embed>`**, **`<frame>`**, **`<iframe>`**, **`<img>`**, **`<object>`**, **`<script>`** and **`<style>`**. This tag deletion cannot be suppressed.

Data of type *multipart/form* may transfer additional parameter attributes for the data together with the actual content. In case of file uploads from Web forms, the original client-side file name is captured in an additional result element named by appending "_filename" to the basic data field name. The **MIME** type of the data, if transmitted, is similarly stored in an extra element with the appended name part "_type".

A single unnamed field in the data string is accepted by the decoder. The data is stored as element *unnamed* in the decoded result.

In case a field name is occurring more than once in a data string, the additional instances are appended to the first instance as list elements. The final content of the decoded field is then a list of all values found in the input.

If an output variable parameter is used, all existing array elements are removed before decoding starts. In case results should be accumulated, the -*noreset* option can be used to suppress the variable reset step.

After the parameter listing fields not to be trimmed an arbitrarily long list of pairs of field names and their default values can be specified. If a field name listed there was not present in the input data, it is set in the result list and array variable with the default value following the field name.

The structure of the arguments is intentionally different in the **PYTHON** version of the command. All arguments after the CGI data must be passed as named arguments. The *noreset* option requires a boolean argument value. Instead of an open argument set for field names and their default after the standard arguments a dictionary with field names as key and defaults as values is used.

Example:

```
URL: http://www.example.com/cgi-bin/doit.tcl?a=alpha&b=beta&b=gamma
```

The data sent by invoking the **CGI** script via this direct **URL** can be decoded in the *doit.tcl* script with the statement

```
uncgi $env(QUERY_STRING) params
```

An array variable named *params* is set up, or reset if already present, and it contains exactly two elements: Element *a* with value “alpha”, and element *b* with value “beta gamma”

unzip

```
unzip nested_list ?index?
unzip(data=,?index=?)
```

Extract a list of indexed elements from a nested list. From each element of the nested input list, the element indicated by the index parameter is selected and appended to the result list. If the index parameter is omitted, the default is zero. If a selected list element does not exist, an empty string is substituted.

Example:

```
set l2 [unzip {{a 1} {b 2}} 1]
```

The `l2` variable is set to a list with elements 1 and 2.

upload

```
upload filename url ?mimetype?
upload(filename=,url=,?mimetype=?)
```

Upload a file to a remote destination. The first argument is a path to a local, readable file. The second argument is an **URL** which indicates the transfer protocol and the remote location. The optional final argument is e **MIME** type specification which is transferred to the remote host in addition to the file content. If it is not specified, the type is implicitly **`application/octet-stream`**.

The supported protocols currently are:

- **`sftp`** and **`ftp`**
  Standard or secure file transfer protocol. The **URL** follows the standard schema
  `ftp://?user?:password?@?host?:port?/?path/??filename?`
  `sftp://?user?:password?@?host?:port?/?path/??filename?`
- Amazon S3
  Store the file in an Amazon **S3** bucket. The **URL** follows the standard schema
  `s3://bucket/?path/?filename`
  Additionally, the **URL** query parameters *secret*, *key*, *acl* and *region* are recognized. If the **AWS** parameters are not specified in the **URL**, the values are taken from **`::cactvs(default_aws_secret)`**, **`::cactvs(default_aws_key)`**, **`::cactvs(default_aws_acl)`** and **`::cactvs(default_aws_region)`**.

The **ACL** can be any of *private*, *public-read*, *public-read-write*, *authenticated-read*, *aws-exec-read*, *bucket-owner-read*, *bucket-owner-full-control* or *log-delivery-write* (see **AWS** documentation). The default is *private*. The available **AWS** regions are frequently updated, please refer to the **AWS** documentation. The specified value is not checked on the client side.

Example:

```
upload mydata.sdf s3://$bucket/folder1/mydata.sdf?region=eu-central-1
```

In this example, the global control variable settings for the **AWS** key, secret and **ACL** are used, while the storage region is explicitly specified.

vec

```
vec add v1 ?v2?...
vec centroid p1 p2 ?p3?...
vec create vspec
vec crossproduct v1 v2
vec degangle v1 v2
vec distance p1 p2 ?p3?...
vec dotproduct v1 v2
vec fitline p1 p2 ?p3?...
vec fitplane p1 p2 p3 ?p4?...
vec len v1
vec len p1 p2 ?p3?...
vec negate v
vec normalize v
vec plane p1 p2 p3 ?p4?...
vec planenormal p1 p2 p3
vec pt_line_distance p1 p2 v2
vec pt_plane_distance p1 d v1
vec radangle v1 v2
vec scale v d
vec subcommands
vec subtract v1 ?v2?...
Vec(vspec)
Vec.Create(vspec)
dir(Vec)
v.add(?v1?,...)
Vec.Add(?v1?,...)
Vec.Centroid(p1,p2,...)
p.centroid(p1)
v.crossproduct(v1)
v.degangle(v1)
Vec.Distance(p1,p2,?p3?,...)
p.distance(p1)
v.dotproduct(v1)
Vec.Fitline(p1,p2,?p3?,...)
Vec.Fitplane(p1,p2,?p3?,...)
Vec.Len(v1)
Vec.Len(p1,p2,?p3?...)
v.len()
p.len(p1,?p2?,...)
v.negate()
v.normalize()
Vec.Plane(p1,p2,p3,?p4?,...)
Vec.Planenormal(p1,p2,p3)
Vec.PtLineDistance(p1,p2,v2)
```

```
p.ptLineDistance(p2,v2)
Vec.PtPlaneDistance(p1,d,v1)
p.ptPlaneDistance(d,v1)
v.radangle(v1)
v.scale(d)
v.subtract(?v1?,...)
Vec.Subtract(?V1?,...)
v + v1
v - v1
v += v1
v -= v1
v * v1
-v
```

This command provides a couple of useful 3D vector and point manipulation commands. In the **Tcl** case, the points and vectors are internally standard **Tcl** list objects with floating point elements, so operations are not blazingly fast but still reasonably effective for routine vector arithmetic. Vectors are not toolkit objects, there is nothing to destroy when they are no longer needed. In the **Python** implementation, there is a custom point/vector object which holds a triple of floating-point values.

This simple utility command only supports vectors and points of 3 dimensions. There is no syntactic difference between vectors and points. The exact interpretation depends on the context.

Vectors can be decoded from a variety of string representations. A vector can be made from

- one of the reserved names *x, y, z, xy, xz, yz* or *xyz*, optionally prefixed by a minus sign. These forms generate a 3D vector of length one pointing into the indicated positive or negative direction.
- By the reserved name *origin* (or *o*). This sets up a 0.0/0.0/0.0 origin point.
- A triple of floating point values.
  The vector components are directly taken from these numbers. No normalization takes place. The triple can be specified either as three numbers, or a string which is parsed.
- A pair of floating point values.
  The z coordinate is set to zero. The *x/y* parts are not normalized.
- A single floating point value.
  All vector components are initialized to that value. No normalization is performed.

The following subcommands are understood:

`vec` *`create arg1 ?arg2 arg3?`*

Create a point or vector and return its fully expanded numerical representation. If only one argument is given, any of the standard representations is accepted. In case three arguments are provided, they are internally concatenated and this list submitted to the decoding procedure. The command result, if the arguments could be decoded, is the numerical vector representation. The vector commands generally understand both resolved vectors, and the short forms which can be decoded by this command, to in many cases, explicit syntax resolution is not required.

```
vec add v1 ?v2...?
```

Add two or more vectors. The return value is the vector sum. Subcommand aliases are **`plus`** and **`+`**. The **PYTHON** implementation provides both class and object methods. Additionally, the **`+`** and **`+=`** numerical operators are overloaded.

```
vec subtract v1 ?v2...?
```

Add two or more vectors. The return value is the vector sum. Subcommand aliases are **`minus`** and **`-`**. The **PYTHON** implementation provides both class and object methods. Additionally, the **`-`** and **`-=`** numerical operators are overloaded.

```
vec degangle v1 v2
vec radangle v1 v2
```

Compute the vector angle between *v1* and *v2* in degrees or radians. The subcommand may be shorted to *deg* or *rad*.

```
vec len v1
vec len p1 p2 ?p3...?
```

If only a single argument is specified, interpret the argument as vector and compute its length. In case multiple arguments are used, compute the length of the straight, acyclic path between the points.

```
vec distance p1 p2 ?p3...?
```

Return a list of the distances between point one and two, two and three, and so on. For two arguments, the result is the same as using **`vec len`**, but if there are additional points, the individual distances between the waypoints are returned as a list instead of the total length of the path.

```
vec centroid p1 p2 ?p3...?
```

These command aliases compute the centroid of the points passed as argument. The return value are the centroid point coordinates. **`midpoint`** is a subcommand alias.

```
vec dotproduct v1 v2
```

Compute the dot product of the two vectors. The result is a floating point value. **`dot`** and **`*`** are subcommand aliases.

```
vec crossproduct v1 v2
```

Compute the vector cross product of the two passed vectors. The return value is the cross product vector. The subcommand may be abbreviated as *cross*, and additionally **`prod`**, **`product`** and **`x`** are subcommand aliases. In Python, the **`*`** multiplication sign is overloaded with this function.

```
vec negate v1
```

Negate the vector. This is equivalent to scaling with minus one.

```
vec normalize v1
```

Normalize vector to length one. The normalized vector elements are the return value. **`unit`** is a subcommand alias.

```
vec scale v1 d
```

Scale length of vector *v1* by the specified floating point factor. The return value is the scaled vector.

`vec fitline` *`p1 p2 ?p3...?`*

Fit a line to a set of points. The return value is a list of a line point and the direction vector.

`vec fitplane` *`p1 p2 p3 ?p4...?`*

Fit a plane to a set of points. The result is returned in Hesse normal form: The closest distance of a plane point to the origin, and the normal vector of the plane.

`vec plane` *`p1 p2 ?p3...?`*

Compute a plane (fitted in case more than three points are specified) spanned by the specified vectors. The result are two orthogonal plane vectors.

`vec planenormal` *`p1 p2 p3`*

Get the normal vector of the plane specified by the three points.

`vec pt_line_dist` *`pt l_point l_vec`*

Compute the distance between a point (coordinates in first argument) and a line in 3D space (specified by an anchor point on the line and the direction vector).

`vec pt_plane_dist` *`pt p_dist p_normalvec`*

Compute the distance between a point (coordinates in first argument) and a plane in 3D space (specified in Hesse form, i.e. smallest distance of any point on the plane to the origin and plane normal vector).

zip

`zip` *`list1 ?listn?...`*

`zip(sequence,?sequence...?)`

Merge one or more lists. The result is a nested list where each element is a sequence of input list elements at the corresponding position. The size of the largest input list determines the output. If any input lists are shorter than another input list, empty elements (**`None`** for **PYTHON**) are substituted.

Example:

```
set l [zip {a b c} {1 2}]
```

The result is a single list „**`{a 1} {b 2} {c {}}`**“

It is possible and sometime useful to use only a single argument list. Here, every element is reformatted so that itself is a proper nested list. Example:

```
set l [zip {a {b c} d}]
```

The result is a list „**`{a {{b c}} d}`**“ with an extra nesting level to protect the middle element so that it can a reclaimed unchanged by un-zipping with a command like

```
set l0 [unzip $l 0]
```

Tcl and Python Environments

When a **Tcl** script is started, the standard set of **Tcl** and, if running with the **Tk** widgets support, **Tk** variables is available in the interpreter.

In addition, the toolkit provides access to internal toolkit status information via the global array variable `::cactvs` in **Tcl**, or a dictionary with the same name in **Python**. It is automatically visible both in the top-level interpreter and in slave and thread interpreters. Additionally, the control variables in the language interpreters are linked, i.e. a change made by **Tcl** is visible in the **Python**, and vice versa.

Many of the elements in the array may be modified by script commands. If they are manipulated this way, their effects are immediate, but not persistent, and usually visible in all interpreters and, if the value is linked to a core library value, throughout the library for compiled code. The exception are specific elements which are thread-local. Changes to these only modify the seen data to all interpreters in that thread, and library code executing in that thread context.

Parameter changes are forgotten the next time the interpreter is executed.

- *cactvs(allowed_read_locations)*
  A set of directories which are accessible to the software as data file input locations. If it is an empty set, no directory access check is performed. All toolkit commands which use file I/O perform check their access rights against this list. This option is primarily useful when, for example, allowing a Web tool to execute external script code. Such a feature should then always be combined with a standard safe **Tcl** interpreter configuration. This is a read-only element. It can be modified by the start-up configuration file.

- *cactvs(allowed_write_locations)*
  A set of directories which are accessible to the software as data file output locations. If it is an empty set, no directory access check is performed. All toolkit commands which use file I/O perform check their access rights against this list. This option is primarily useful when, for example, allowing a Web tool to execute external script code. Such a feature should then always be combined with a standard safe **Tcl** interpreter configuration. This is a read-only element. It can be modified by the start-up configuration file.

- *cactvs(application)*
  Short application name.

- *cactvs(application_build_date)*
  The build date of the main program or interpreter, in seconds since 1970, suitable for use with `clock format`.

- *cactvs(application_long_name)*
  Long name of application.

- *cactvs(application_mode)*
  A code for the type of application context the **Tcl** interpreter is operating in. Possible values are 0 (undefined), 1 (simple standalone executable), 2 (executable running encapsulated **Tcl** script), 3 (full interactive interpreter), 4 (restricted interactive interpreter), 5 (full interpreter

running command script), 6 (restricted interpreter running command script), 7 (link library for 3rd party application), 8 (TCL module for 3rd party TCL interpreter), 9 (TCL module for TCL Web browser plug-in), 10 (PYTHON module with secondary TCL partner interpreter for 3rd party PYTHON interpreter) or 11 (database UDF module).

- *cactvs(aromaticity_model)*
  The aromaticity model to use. Possible values are *cactvs* (the default), *daylight* (which has some nasty problems, but using it will improve compatibility of the results of `Smarts` matching with Daylight software, and *tripos*, for a really weird and not very useful system.

- *cactvs(async_computation_receiver_port)*
  Port number used for asynchronous networked property computation requests.

- *cactvs(auto_container_creation)*
  If set, wrapper containers implied by property data retrieval statements on potential container element objects are automatically created. By default, such a data query fails if the container does not yet exist. Examples are the retrieval of reaction property data from an ensemble (if the ensemble is not yet part of a reaction, a reaction is created, and the ensemble becomes the reagent component), or dataset property data from any object which could be a dataset element (ensembles, reactions, tables, networks, but not datasets themselves, here the query accesses the data content of the original dataset).

- *cactvs(auto_exit_timeout)*
  The spent CPU time in milliseconds after the program automatically terminates. If the value is zero, no timer is set.

- *cactvs(auto_vector_extension)*
  A boolean flag which controls the behavior of the toolkit when individual elements of vector property data are set with an element index that is larger than the current size of the vector. By default, only existing vector indices (`0` to `$size-1`) and the new element just beyond current vector size (`$size`) may be used. If the flag is set, any vector index may be addressed. Missing vector elements between the newly set element and the current vector size are filled with default values.

- *cactvs(auxdatapath)*
  Search path for auxiliary data files required by computational modules.

- *cactvs(base_os)*
  The operating system core version, i.e. something like *Linux 2.6* or *Linux3.11* for Linux distributions. This is a read-only string.

- *cactvs(biologics_count)*
  The current number of biologics objects in memory. This is a read-only attribute.

- *cactvs(biologics_table)*
  The handle of the table with biologics fragment expansion data (usually this is *table3*). This attribute is read-only.

- *cactvs(biologics_support)*
  Read-only boolean flag indicating whether this toolkit version has been compiled with biologics object support.

- *cactvs(bmp_support)*
  Read-only boolean flag indicating whether this toolkit version support image generation in various legacy Windows bitmap formats.
- *cactvs(bunzip2_program)*
  The fully resolved path name of the *bunzip2* program used for I/O of bz2-compressed files.
- *cactvs(byte_order)*
  A read-only element informing about the byte order of the current platform. It is set to to either *littleendian* or *bigendian*.
- *cactvs(cache_lifetime)*
  CACTVS maintains a cache in the user's home directory to cache files obtained from databases and Internet **URL**s. This parameter defines the lifetime of objects in the cache in seconds. Older files are ignored and searched again via their type-specific standard search path.
- c*actvs(captcha_image_file)*
  The name of the file with last **CAPTCHA** image received as result of an attempt to contact an Internet service. This can happen for example when up- or downloading a Google Docs spreadsheet table file. If no such file exists, the element is an empty string. The file is also deleted and the variable is reset when a decoded **CAPTCHA** was successfully used for log-in.
- *cactvs(captcha_string)*
  The decoded contents of the **CAPTCHA** image file above. Its contents are used when attempting to access protected Internet sites. This variable must be set with user interaction. By default, it is a `NULL` string.
- *cactvs(captcha_token)*
  The access token associated with the **CAPTCHA** image file and decoded string. This is used as credential for Internet service log-ins. The variable is typically set together with the **CAPTCHA** image file when a service requires additional authentication. Scripts would then present the image, get the user to decode it, store the decoded string in the `cactvs(captcha_string)` variable and then attempt another log-in. Modules which support **CAPTCHA** log-ins automatically read and set the **CAPTCHA** variables in the course of their operation.
- *cactvs(cmdxpath)*
  Search path for **TCL** command extension modules. The standard **TCL** package path is automatically added to this search path. The main difference between a **CACTVS TCL** expansion and a standard **TCL** package is the presence of a descriptor structure with metadata. **CACTVS TCL** extensions may be loaded as normal **TCL** packages, but then do not provide access to the module description data via the `cmdx` command.
- *cactvs(color_support)*
  A read-only boolean flag indicating if the decoding of named colors is supported, either by access to a local **X11** color names database or by a compiled-in database.
- *cactvs(compatible_os_variants)*
  Read-only regular expression used to match compatible OS variants for dynamically loading modules.

- *cactvs(compress_support)*
  A read-only boolean flag indicating that the processing of compressed data is supported. This includes *zlib* and *gzip* compression and on most platforms *bzip2*. Internally, some data file formats use the *lzo* compression format.
- *cactvs(connect_timeout)*
  The maximum number of seconds to wait and, if necessary, retry after a short break in the process of establishing a socket connection. If set to zero, only a single connection attempt is made. Setting it to a negative value configures an unlimited number of connection attempts - but then a command trying to connect to a dead service never returns.
- *cactvs(corina_support)*
  A boolean read-only flag indicating whether the application was build with an integrated **CORINA** module for computing atomic 3D coordinates (`A_XYZ`, `A_CONFORMER`). **CORINA** support is limited to special toolkit editions and not contained in normal academic or commercial toolkit packages.
- *cactvs(cpu_limit)*
  A read-only variable describing any hard CPU execution time limits set for the interpreter. If the value is -1, which is the default, no limits have been set. Setting a CPU limit is only supported for specific interpreters on certain platforms (for example, *tclcactvs* on Linux), and limits must be set at start-up (option -C for *tclcactvs*).
- *cactvs(crypt_support)*
  A read-only boolean flag indicating whether the application support encryption by the crypt (and other) algorithms.
- *cactvs(currency_converter_host)*
  The default currency exchange rate host. If set to an empty string, Internet-based currency conversion is disabled. Conversion is automatically run if a monetary value is requested in a different currency from the one the data is stored in, and the exchange rate has not yet been cached. Example:
  ```
  ens set $eh E_PRICE "100.50 EUR"
  ens get $eh E_PRICE(USD)
  ```
- *cactvs(custom_config_file)*
  The name of a custom toolkit configuration file specified as command line argument. It is an empty string if no such file was set. The value is read-only.
- *cactvs(custom_python_attributes)*
  A modifyable value which determines whether **Python** wrapper objects of core toolkit objects may store additional custom attributes, in addition to the standard toolkit-implemented attributes. By default this flag is not set in order to catch errors in the naming of attributes. If set, any attribute that is not a standard attribute is stored in a private object dictionary, without rasing an error.
- *cactvs(data_directory)*
  The name of the installation directory which contains - mostly in well-known subdirectories - script libraries, loadable modules, filters, property definitions, images, help files and other auxiliary files needed for the proper operation of the toolkit. The stand-alone interpreter *csweb* (and its variants) are the only toolkit applications which do not access this resource.

- *cactvs(database_file_key)*
  An integer indicating that for access to data files a suitable file key must be present. The default value is zero, meaning that no such key is needed. This is only used in very special circumstances when a Cactvs library was built to allow operation only on specifically named files.
- *cactvs(database_log_file)*
  The name of a file to write logging information for database accesses to.
- *cactvs(database_log_mode)*
  An boolean value indicating the current log mode. Disabled logging is represented by value 0.
- *cactvs(databasepath)*
  Search path for database connector modules.
- *cactvs(dataset_count)*
  The current number of dataset objects. Note that not all datasets necessarily have **Tcl** handles or **Python** wrapper objects, so there may be a discrepancy between this value and the result list length of the `dataset list` command.
- *cactvs(dataset_swap_threshold)*
  Maximum size of datasets before disk swapping of contained objects is automatically performed. The default value is 10000. A negative value disables automatic disk swapping.
- *cactvs(datatool_program)*
  The name or, if not found in a standard location, fully specified path of the **NCBI** *datatool* program. This program is needed to process PubChem **ASN.1** data in text format. For binary **ASN.1** data, this program is not used.
- *cactvs(db_support)*
  A read-only boolean flag indicating whether this interpreter supports database access functions and loadable database client modules (`dbase` and `dbx` **Tcl** commands).
- *cactvs(default_aws_acl)*
  The default file **ACL** for newly uploaded **AWS** files. A null value means it is unset.
- *cactvs(default_aws_key)*
  The default public key string for accessing **AWS** bucket store as identified user. A null value means it is unset.
- *cactvs(default_aws_region)*
  The default region for newly uploaded **AWS** files. A null value means it is unset.
- *cactvs(default_aws_secret)*
  The default local secret for encrypting access parameters for the **AWS** bucket store. A null value means it is unset.

- *cactvs(default_currency)*
  The default currency, which is implicitly assumed when dealing with currency data without an explicit currency indicator. The string should be set to the standard international trade name of the selected currency (i.e. EUR, USD, JPY) though a couple of colloquial names are also recognized (Euro, US$, Yen). In any case, a maximum of four characters are significant.

  The attribute is thread-local. Changing it has sets the default for future threads only if the modification is performed in the base thread, otherwise these inherit the current setting in the base thread.
- *cactvs(default_database)*
  Name of the default database. These default values are used to initialize *dbx* database interface objects and do not directly influence operation.
- *cactvs(default_database_host)*
  Default database host.
- *cactvs(default_database_options)*
  Default database options to use in the connection string. The format is database-dependent.
- *cactvs(default_database_password)*
  Default database access password.
- *cactvs(default_database_type)*
  Type of the default database connector (*mysql*, *oracle*, etc.), normally *mysql*.
- *cactvs(default_database_user)*
  Default database user.
- *cactvs(default_element_count_property)*
  The property used by default for getting element counts for formula matching, usually `E_ELEMENT_COUNT`.
- *cactvs(default_http_agent)*
  The default agent string sent to Web sites when using http-based communication. By default, it uses the string of a reasonably modern Firefox browser.
- *cactvs(default_hydrogen_addition_mode)*
  The default hydrogen processing mode for file objects. Possible values are *asis*, *add*, *strip*, *stripall*, and *addblind*. More information can be found in the documentation of the **`molfile set`** command in the section describing the *hydrogens* attribute.
- *cactvs(default_license)*
  The default license class for newly created objects which have a license class field. It can be one of *xemistry, gpl2, gpl3, lgpl, apache, creativecommons, publicdomain, proprietory, confidential, classified* or *usgovwork.* For commercial toolkit versions, the default is *proprietary*, and for US government clients *usgovwork.*
- *cactvs(default_reaction_isotope_hash_property)*
  The default property for isotope- but not stereo-specific reaction hash codes (usually `X_ISOTOPE_HASH128`)

- *cactvs(default_reaction_isotope_stereo_hash_property)*
  The default property for stereo- but not isotope-specific reaction hash codes (usually `X_ISOTOPE_STEREO_HASH128`)
- *cactvs(default_reaction_simpleo_hash_property)*
  The default property for stereo- and isotope-specific reaction hash codes (usually `X_HASH128`)
- *cactvs(default_reaction_stereo_hash_property)*
  The default property for stereo- but not isotope-ignorant reaction hash codes (usually `X_STEREO_HASH128`)
- *cactvs(default_reaction_screen_property)*
  The default property to use for reaction query screening. This is typically `X_SCREEN`.
- *cactvs(default_similarity_property)*
  The default property to use for similarity queries. This is typically `E_SCREEN`.
- *cactvs(default_spinach_formula)*
  Thew default formula for matching EliLilly-inspired *spinach*-class query superatoms indicating a small structural decoration. The initial value is **`">=C0-2[Het]0-2"`**, i.e. zero to two carbons, zero to two hetero atoms, no hydrogen specification. Larger substituents do not qualify as spinach.
- *cactvs(default_structure_isotope_hash_property)*
  The default property for isotope- but not stereo-specific ensemble hash codes (usually `E_ISOTOPE_HASH128`)
- *cactvs(default_structure_isotope_stereo_hash_property)*
  The default property for stereo- but not isotope-specific ensemble hash codes (usually `E_ISOTOPE_STEREO_HASH128`)
- *cactvs(default_structure_simpleo_hash_property)*
  The default property for stereo- and isotope-specific ensemble hash codes (usually `E_HASH128`)
- *cactvs(default_structure_stereo_hash_property)*
  The default property for stereo- but not isotope-ignorant ensemble hash codes (usually `E_STEREO_HASH128`)
- *cactvs(default_substructure_screen_property)*
  The default property to use for substructure query screening. This is typically either `E_SCREEN` or `E_QUERY_SCREEN`.
- *cactvs(default_superstructure_screen_property)*
  The default property to use for superstructure query screening. This is typically either `E_NO_HYDROGEN_SCREEN` or `E_NO_HYDROGEN_QUERY_SCREEN`.
- *cactvs(default_system_encoding)*
  The character encoding used by the environment. It is typically **UTF-8** or **ISO8859-1**
- *cactvs(default_tauto_hash_property)*
  The standard tautomer structure hash property (usually `E_TAUTO_HASH128`)

- *cactvs(default_warmup_reaction)*
  A line encoding of the default reaction to use with the **`prop use`** command if a property does not have its own specific computation warm-up reaction definitions.
- *cactvs(default_warmup_structure)*
  A line encoding of the default structure to use with the **`prop use`** command if a property does not have its own specific computation warm-up structures. The standard structure contains aliphatic and aromatic rings, multi-ring ringsystems, atom and bond stereochemistry, aromatic and aliphatic pi systems, an isotope label and other structural attributes which might play a role in property computation.
- *cactvs(dependency_warning)*
  If set, warn about property dependencies detected when parsing a property definition which involves dependent properties which are not yet defined. Having such dependencies is not illegal, and the dependencies are updated at a later time when the unresolved, referred property has been set up. However, in case these properties are never read, dangling dependencies are never resolved and ignored, and this may indicate a programming error.
- *cactvs(disable_io_modules)*
  If set to 1, no file format extension modules are loaded, even if they are listed in the local system configuration file for automatic loading. Also, automatic look-up for extension modules on file suffixes is blocked.
- *cactvs(disable_toolkit_rpc)*
  If this flag is set, **RPC** communication with other **CACTVS** applications is disabled. Since the Windows port currently does not support **RPC** inter-process communication, this flag has no effect on that platform. This flag only applies only to internal **RPC** between toolkit applications, not to operating the interpreter as **KNIME RPC** server.
- *cactvs(distribution)*
  The platform this package was compiled for, including distribution information. Example: *Linux2.6-SuSE10.2-64*. This element is read-only.
- *cactvs(do_database_lookup)*
  If set, look-up of extension modules in databases is enabled.
- *cactvs(do_signals)*
  If set (which is the default), untrusted or flaky property computation modules may be executed with a set of signal handlers in place, which try to recover from segmentation fault errors and other signals. Recovery is not guaranteed to work if the module corrupted working memory. The signal handler overhead is not insignificant, and properties should be configured to rely on installed signal handlers only if required, and where the problem is not fixable, for example because the source code is not available.
- *cactvs(do_timeouts)*
  If set, allow time-outs in property computations to occur. This flag is on by default.

- *cactvs(dynamic_loader_mode)*
  If this flag is to a positive value (it is set by default), the toolkit allows the loading of dynamic extension modules of all supported classes if not prohibited by class-specific flags. Specifically, a value of one allows loading of modules found in the trusted path, a value of two loading from both the standard and trusted path, with the trusted path having preference, and a value of three again loading from both paths, but the the standard path having preference.
- *cactvs(element_sequence)*
  A list of the numbers of the elements of the periodic system in standard Hill formula sequence (6 = carbon, 1=hydrogen, 0 = pseudo-atoms, 89= Actinium, and so forth).
- *cactvs(element_table)*
  The handle of the table with element data (usually this is *table0*). This attribute is read-only.
- *cactvs(ens_count)*
  The current number of ensemble objects. Note that not all ensembles necessarily have **TCL** handles or **PYTHON** wrapper objects, so there may be a discrepancy between this value and the result list length of the `ens list` command.
- *cactvs(eol_chars)*
  The platform-dependent default end-of-line character(s).
- *cactvs(eutils_contact)*
  The contact email to include in **NCBI EUTILS** queries. By default, it is a generic *xemistry.com* address.
- *cactvs(eutils_host)*
  The name of the host with the **NCBI EUTILS** suite of **ENTREZ** database access services. If set to an empty string, **ENTREZ** access is disabled. Setting it to a different host is only useful if you run an **ENTREZ** clone in-house. The default is *eutils.ncbi.nlm.nih.gov.*
- *cactvs(eutils_toolname)*
  The tool name to report when submitting **NCBI EUTILS** queries. The default is *CactvsToolkit*.
- *cactvs(executable_type)*
  The platform model the executable was compiled for. It is either 32 or 64 bits. This information is read-only.
- *cactvs(executablepath)*
  Search path for executable external programs, such as compressors and decompressors, which in some circumstances are automatically launched for I/O operations.
- *cactvs(explicit_stereo_h)*
  A global formatting flag used by various property computation routines and file I/O modules which indicates preference to retain hydrogens at stereo centers for maximum interpretation reliability, or whether to strip them and to rely on the proper re-attachment by a reader or decoder. By default this flag is set, i.e. stereo hydrogen is preferentially retained.
- *cactvs(expr_support)*
  A read-only boolean flag which indicates whether the application supports **SQL**-style expressions on data attached to objects, for example via the `expr` object commands.

- *cactvs(extension_support)*
  A read-only boolean flag which indicates whether this interpreter is capable of loading extension modules, for example file I/O format handlers or datatype handler modules.
- *cactvs(external_colordb)*
  If set, the software tries to preferably access a locally installed **X11** color database, overriding the built-in color collection. The internal color set is used as fall-back is the flag is not set, or no **X11** color database can be found.
- *cactvs(extra_format_flags)*
  A list of the names of extra formatting bits which should be used for property data formatting in the interpreter. By default this is an empty list.
- *cactvs(fcgi_redirect)*
  This read-only boolean variable is an indicator on whether the *stdout*, *stderr* and *stdin* standard I/O channels are currently redirected to the corresponding **FCGI** channels. To activate or cancel redirection, use the **fcgi** **TCL** command, which is available in some interpreter versions (*tclcactvs*, *csweb*).
- *c actvs(fcgi_support)*
  A read-only boolean flag indicating whether this application was compiled with **FCGI** (fast **CGI**) support. This flag informs about the availability of the **fcgi** script command as well as the ability of various output mechanisms to directly stream into an **FCGI** channel, or a standard channel redirected into a **FCGI** channel.
- *cactvs(filexpath)*
  The search path for structure and reaction file I/O module extensions
- *cactvs(filterpath)*
  The search path for filter definitions.
- *cactvs(fontpath)*
  The search path for TrueType fonts used by several imaging modules.
- *cactvs(full_smarts_support)*
  A read-only boolean flag indicating whether this interpreter was compiled with a full **SMARTS** parser. **SMILES** and some basic **SMARTS** (essentially the flat match attribute level as found in query *Molfiles*) can be processed even if this capability was not added.
- *cactvs(gdbm_support)*
  A read-only boolean flag indicating whether this interpreter was compiled with **GDBM** key/value store support.
- *cactvs(gunzip_program)*
  The fully resolved path name of the *gunzip* program used for I/O of *gzip*-compressed files. An external decoder is only forked if the file is a simple file, and might contain formatting features which cannot be resolved by the simple zlib-based internal program emulation.
- *cactvs(gzip_program)*
  The fully resolved path name of the *gzip* program used for I/O of *gzip*-compressed files. An external decoder is only forked if the file is a simple file, and might contain formatting features which cannot be resolved by the simple zlib-based internal program emulation.

- *cactvs(host)*
  The name of the computer the interpreter is running on. Read only.
- *cactvs(host_domain)*
  Domain name of the host. This is often not automatically identifiable, but can be set by scripts.
- *cactvs(host_ip)*
  The IP address of the computer the interpreter is running on. Read-only. Depending on the number of network interfaces detected, additional read-only entries named *host_ip2* etc. may be present, including IP address 127.0.0.1 for *localhost*.
- c*actvs(https_enforcement)*
  If set, use *https* instead of *http* for external communication where possible.
- *cactvs(iconv_program)*
  The fully resolved path name of the *iconv* program used for I/O of files encoded in **UCS-2** and other non-Latin1 encodings of text data.
- *cactvs(inchi_support)*
  A read-only boolean flag indicating whether this interpreter supports operations with InChI strings, including decoding of such strings in the **`ens create`** command, computing the `E_INCHI, E_STDINCHI, E_INCHIKEY` and `E_STDINCHIKEY` properties, and I/O support for InChI files with or without auxiliary data.
- *cactvs(internet_lookup_level)*
  This element controls the extent to which Internet-based operations can be implicitly executed. If set to level 0, no external Internet communication takes place. Level one enables the use of Internet services in computational modules and file I/O modules. They are still subject to fine-grained control from the settings of the host control variables and the structure security flag. Level 2 additionally enables the lookup of property definitions and extension modules in Internet repositories named in the search paths. The default Internet lookup level is one.
- *cactvs(internet_support)*
  A read-only boolean flag informing whether this interpreter was compiled with support for accessing Internet resources, for example via **URL**s or **SOAP** messages.
- *cactvs(interrupted)*
  A thread-local flag set when an interrupt (user pressing ctrl-c, time-out of operation with time limit) occurred. Commands which can be interrupted reset the variable to 0 when the command execution begins. Examples: The **`molfile scan`** and **`ens transform`** commands.
- *cactvs(io_support)*
  A read-only boolean flag indicating whether this interpreter supports file-based chemistry data I/O. This flag is only unset in a few specialized link libraries where chemistry data is directly stored in the internal data structures by a third-party application program.
- *cactvs(is_slave)*
  A read-only boolean flag indicating whether this interpreter is a slave interpreter (for example, a property computation interpreter for scripted properties) or not. Thread main interpreters are not slaves.

- *cactvs(itcl_support)*
  A read-only boolean flag indicating whether the **Itcl Tcl** object-oriented programming extension commands are available in this interpreter as a compiled-in component. If not, the Itcl module may still be loaded with standard **Tcl** *load* or *package* commands.
- *cactvs(java_class_path)*
  The Java VM class path, if this is an executable which also has a Java VM compiled in.
- *cactvs(java_error_message)*
  The last error message generated by a **Java** tool.
- *cactvs(json_support)*
  A read-only boolean flag indicating whether the application was compiled with an integrated **JSON** parser. This flag informs about the availability of the `json` script command, and support for various json-based encoding and decoding schemes, for example for table data I/O.
- *cactvs(keyxpath)*
  The database key extension module search path.
- *cactvs(kill_object_limit)*
  The maximum number of toolkit objects to be present in memory at any time before the program auto-terminates in order to avoid memory hogging. This is primarily a safety feature to prevent scripts which leak objects from wrecking havoc. A negative value indicates that no limit is enforced.
- *cactvs(knimenode_deadman_timeout)*
  The number of milliseconds a **KNIME** node function may execute in a single invocation before the application is terminated to protect the server integrity. If this attribute is used, it should be set to a somewhat larger value than the soft node invocation timeout. A value of zero or less disables the deadman timer.
- *cactvs(knimenode_debug)*
  If set to a true value, **KNIME** nodes are run in debug mode.
- *cactvs(knimenode_default_editurl)*
  If compiled with **KNIME** node support, the **URL** of an Internet-accessible node editor application such as *https://xemistry.com/NodeDesigner.*
- *cactvs(knimenode_default_host)*
  If compiled with **KNIME** node support, the name of the default **RPC** host compiled into nodes.
- *cactvs(knimenode_default_password)*
  If compiled with **KNIME** node support, the default **RPC** password. If the program is run as a **KNIME** node **RPC** server, this is the password external node users must provide to gain access if no specific user/password combinations are configured.
- *cactvs(knimenode_default_port)*
  If compiled with **KNIME** node support, the default **RPC** port compiled into nodes. The default port is 16570.

- *cactvs(knimenode_default_rpc_logfile)*
  The name of a file to log the **RPC** communication between a node server and a **KNIME** workbench.
- *cactvs(knimenode_default_user)*
  If compiled with **KNIME** node support, the name of the default **RPC** user. If the program is run as a **KNIME** node **RPC** server, this is the user name external node users must provide to gain access if no specific user/password combinations are configured.
- *cactvs(knimenode_fork)*
  If the program is run as **KNIME RPC** server, and this flag is set, individual **KNIME** nodes are executed on forked server application instances instead of a shared executable. Nodes which use a **PYTHON** interpreter are always forked, due to **PYTHON** multi-threading limitations.
- *cactvs(knimenode_max_exectime)*
  If the program is run as **KNIME RPC** server, this is the maximum wall time in seconds allowed to be spent in the execution script per node invocation. If the time is exceeded, the script execution is stopped and an error reported to the workbench.
- *cactvs(knimenode_max_file_size)*
  If the program is run as **KNIME RPC** server, this is the maximum size of transferred file blobs (for example, reading upload files which are not accessible on the local file system). A negative value disables the check.
- *cactvs(knimenode_max_rows_per_port)*
  If the program is run as **KNIME RPC** server, this is the maximum number of rows accepted or sent per table port per node invocation.
- c*actvs(knimenode_max_rows_total)*
  If the program is run as **KNIME RPC** server, this is the maximum number of rows combined over all table-type input and output ports of a node which are accepted and sent per node invocation across all ports.
- *cactvs(knimenode_max_statements)*
  If the program is run as **KNIME RPC** server, this is the maximum number of **TCL** statements which may be executed per node invocation. If this number is exceeded, script execution is terminated and an error reported.
- *cactvs(knimenode_safe_interpreters)*
  If set, all **KNIME** nodes are executed in safe interpreters, which limits the type of script statements they can use.
- *cactvs(knimenode_support)*
  A read-only flag indicating whether this interpreter was compiled with support for working with **KNIME** nodes (`knode` command).
- *cactvs(largefile_support)*
  A read-only boolean flag indicating if this version of the library supports 64-bit file operations. There are probably no platforms that are still supported which do not allow operations on files with more then 2 or 4GB.

- *cactvs(lhasa_support)*
  A read-only boolean flag indicating that the interpreter supports the *lhasa* object which understands **Lhasa** reaction transforms and the parsing of **Patran** and **Chmtrn** data.

- *cactvs(library_compile_date)*
  The read-only compilation date of the main toolkit library in seconds since 1970. This value is suitable for use with `clock format`.

- *cactvs(library_compile_flags)*
  The C compiler flags used for compiling the main toolkit library. The value is read-only.

- *cactvs(library_compiler)*
  The compiler which compiled the main toolkit library. The value is read-only.

- *cactvs(library_directory)*
  The path of the directory where the bundled link libraries for this installation are stored. For stand-alone applications, this element is empty.

- *cactvs(library_error_code)*
  The numerical error code of the last **Cactvs** library error of the current thread. Custom errors, for example raised by modules which use a custom message and not the standard message table have an error code of minus one. This is a read-only element, and thread-local.

- *cactvs(library_error_message)*
  The last raw **Cactvs** library error message of the current thread, without any formatting for the **Tcl** interpreter. Most library messages also generate a **Tcl** error message, but there are many error messages in the toolkit related, for example, to command syntax problems which are part of the scripting layer and not handled by the core library. Such errors do not change the library message. This is a read-only element, and thread-local.

- *cactvs(license_comment)*
  Comments regarding the license. Read only.

- *cactvs(license_developer)*
  Name of the developer of an application. Read only.

- *cactvs(license_hostname)*
  Name of licensed host, empty if not set. Read-only.

- *cactvs(license_hostid)*
  Host-ID of licensed host, 0 if not set. More elements *license_hostid2* to *license_hostid10* are present if multiple host IDs are covered by a license. Since academic packages are not bound to host IDs, this array element is not present in such packages. If present, this is a read-only element.

- *cactvs(license_ip)*
  IP-address for licensed host, 0 if not set. More fields *license_ip2* to *license_ip10* are provided if multiple IP addressed are covered by a license. Since academic packages are not bound to IP addresses, this array element is not present in such packages. If present, this is a read-only element.

- *cactvs(license_maxversion)*
  Maximal toolkit version covered by license. 0 if not set.
- *cactvs(license_minversion)*
  Minimal toolkit version covered by license. 0 if not set.
- *cactvs(license_netmask)*
  Network mask for license. 0 if not set. Read only.
- *cactvs(license_network)*
  Network address for license. 0 if not set. Read only.
- *cactvs(license_serial)*
  Serial number of license. 0 if not set. Read only.
- *cactvs(license_support)*
  A read-only boolean flag indicating whether this interpreter support license handling.
- *cactvs(license_timeout)*
  Time-out date for license in standard Unix notation (seconds since 1970). 0 for permanent licenses. Read only. Use the **`clock format`** command to convert it into a readable representation.
- *cactvs(license_type)*
  License type (*commercial*, *academic*, *government*, *evaluation*). Read only.
- *cactvs(license_user)*
  Licensed host account user name. The element is not present if not set. More elements *license_user2* to *license_user10* are present if multiple user names are covered by a license. Since academic packages are not bound to users, this array element is not present in such packages. If present, this is a read-only element.
- *cactvs(licensee)*
  Licensee of the basic toolkit. Read only. This is a free-form string, not a host account name (see above).
- *cactvs(licensor)*
  Licensor name. Read only.
- *cactvs(lookup_hosts)*
  A list of the names of the hosts used for lookup operations (such as chemical name and **CAS** number resolution in the **`ens create`** command). The default host list is *cactus.nci.nih.gov* (**NCI** resolver) and *opsin.ch.cam.ac.uk* (systematic name resolver). *www.genome.jp* (**KEGG**) and *chemspider.com* are also supported as an additional resolvers and can be added or replaced if desired.

  If this array element is set to an empty string, compound name lookup/resolution is disabled. Operations involving contacting the name lookup hosts are not controlled by the **`::cactvs(structure_security)`** variable setting, because no connection tables are transmitted.

  It is possible to attach a directory/file part to the host name for the **NCI** resolver. If a lookup host is specified in this fashion, the default path (*/chemical/structure/*) is overridden. This feature allows access to experimental releases of this service.

- *cactvs(lookup_mode)*
  This array element can be used to fine-tune the operation of the **NCI** resolver. By default, this element is unset, and the default resolver mode is selected when the resolver host is contacted. One useful supported value is *name_pattern*, which invokes substring name search on the resolver.
- *cactvs(lookup_timeout)*
  The thread-local maximum number of seconds to wait for an answer from an Internet-based structure identifier resolver (such as PubChem for CIDs and SIDs, or KEGG for KEGG IDs). A zero value indicates no timeout.
- *cactvs(lookup_timeout_occurred)*
  This thread-local flag is cleared before the auxiliary **Tcl** or **Python** commands `fetch` or `post` are executed, or an attempt is made to resolve a structure identifier (see `ens create` command) by contacting a remote server. If any of these operations fail due to a server timeout, but not other causes, this flag is set and remains set until the next look-up operation is performed, or the flag is explicitly reset in a script.
- *cactvs(lzo_support)*
  A read-only boolean flag indicating whether the interpreter supports the **Lzo** compression/decompression scheme. This method can for example be used in Minimol compression, resulting in a somewhat faster decompression compared to standard **Zlib**.
- *cactvs(malloc_bytes)*
  Total count of allocated bytes. Only updated when memory debugging is active.
- *cactvs(malloc_count)*
  Count of memory allocation operations. Only updated when memory debugging is active.
- *cactvs(malloc_free)*
  Count of memory release operations. Only updated when memory debugging is active.
- *cactvs(malloc_total)*
  Currently allocated memory byte count. Only updated when memory debugging is active, and only valid on Windows (where you can obtain the original allocation size of a memory block from its address).
- *cactvs(max_scan_threads)*
  The maximum number of scan threads on a single handle which can concurrently use the library. This is a read-only value. Earlier toolkit versions also had limitations on the maximum numbers of library and **Tcl** threads - these are now unlimited.
- *cactvs(maximum_ring_count)*
  The maximum number of rings allowed in an ensemble. This value is primarily intended to act as a safeguard against the processing of structures with insane numbers of rings, as it can happen while decoding overlapping 3D structures, and not as a principal limitation of the toolkit. The default value of 1000 is certainly sufficient even for large proteins.
- *cactvs(maximum_ring_size)*
  The number of atoms in the largest rings which will be detected. The standard value is 80.

- *cactvs(maximum_stereo_distance)*
  The maximum number of bonds traversed to check stereogenicity, and, indirectly, compute various stereo descriptors. Ligands which differ only beyond this bond distance are not considered different for the purpose of stereo handling. The default value is 22, which corresponds to the maximum required value to handle all entries in the FDA registry.
- *cactvs(memory_limit)*
  A read-only variable indicating hard memory consumption limits in bytes which have been set for the application. In case this value is -1, no limits are set, which is the default. Memory limits can only be set for certain interpreter versions (for example, *tclcactvs* on Linux), and only on start-up (option -M for *tclcactvs*).
- *cactvs(minimol_support)*
  A read-only boolean flag which indicates whether this interpreter has support for Minimols.
- *cactvs(mmap_buffer_size)*
  The maximum length of the memory map buffer for field-based file I/O, in bytes. This attribute affects only a few file formats which can utilized large memory buffers, most notably the *cbs* search file format. If the data size of a field exceeds this limit, the data are read by means of traditional file I/O and not consume system memory resources.
- *cactvs(mmap_high_threshold)*
  Maximum size of files which should use memory mapping for accelerated reading. Note that this limited applies only to full file mapping - file formats with direct access query fields will still map those fields for accelerated access.
- *cactvs(mmap_low_threshold)*
  Minimum size of files read via memory mapping for accelerated reading.
- *cactvs(molecule_bond_set)*
  A bit set of those bond types which contribute to grouping atoms into molecules. The default set consists of the bond types *normal, dative, complex, ionic* and *3center.*
- *cactvs(molfile_count)*
  The number of currently active *molfile* object handles. Note that not all structure file access objects necessarily have **Tcl** handles or **Python** wrapper objects, so there may be a discrepancy between this value and the result list length of the `molfile list` command.
- *cactvs(multithreaded_script_support)*
  A read-only boolean flag indicating whether this interpreter supports multi-threaded scripts. This is not equivalent to the *thread_support* flag. The latter is a prerequisite and indicates that the core library supports multiple threads.
- *cactvs(namespace)*
  The local URN namespace, which is for example used during property look-up. By default, it is set to *.local.*
- *cactvs(namespacepath)*
  The namespace resolution path.

- *cactvs(network_count)*
  The current number of network objects. Note that not all networks necessarily have **Tcl** handles or **Python** wrapper objects, so there may be a discrepancy between this value and the result list length of the `network list` command.

- *cactvs(network_support)*
  A read-only boolean flag indicating whether this application was compiled with support for the `network` object, including its `vertex` and `connection` minor objects, plus the associated script commands of the same name.

- *cactvs(netxpath)*
  The search path for network file I/O modules.

- *cactvs(nitrogen_valence_check)*
  The default mode for checking nitrogen valences with the `ens/mol/atom valencecheck` commands. The default is ionic, meaning that pentavalent nitrogen is rejected. The possible values are *xionic*, *ionic*, *asis*, *pentavalent* and *xpentavalent*.

- *cactvs(object_auto_scope)*
  This thread-specific flag controls whether major objects with **Tcl** handles (*ensembles*, *reactions*, *datasets*, *molfiles*, *tables* and *networks*) which are generated within a thread interpreter or slave interpreter are automatically scoped to that interpreter (see ::`cactvs(object_scope)`), meaning that their handle can only be resolved by that interpreter.

- *cactvs(object_scope)*
  This flag controls whether object handles can be resolved in all **Tcl** interpreters of an application or can be limited to an interpreter which claims them and hides them from global visibility. By default, all objects with handles are globally resolvable, regardless of their creation mode. The main purpose of this flag is that is allows scripted computation modules to hide objects (such as substructure ensembles) which they set up once and which persist after the computation finishes. Hidden objects still count for the global object counts (such as `::cactvs(ens_count)`), but, for example, no longer appear in `ens list` executed in a main interpreter when they are created and hidden in a property slave interpreter. This makes the detection of object leaks simpler.

- *cactvs(objectpath)*
  Search path for compiled objects used by property computation modules, including dynamic shared objects, OSX bundles or Windows DLLs.

- *cactvs(ocrhost)*
  The host to use when performing chemical structure drawing **OCR**, for example when reading **GIF** or **PNG** files as chemical structure files where these image do not possess hidden embedded structure data. The default host *cactvs.nci.nih.gov* runs the **OSRA** service.

- *cactvs(online)*
  Flag indicating whether the local host has Internet access. Automatically updated.

- *cactvs(openphacts_id)*
  The application ID for accessing **OpenPhacts** data.

- *cactvs(openphacts_key)*
  The key for accessing **OpenPhacts** data.
- *cactvs(os)*
  The name of the operating system the interpreter is running on, in the full nomenclature used also to name the toolkit packages, i.e. with the name of the operating system distribution. Read only.
- *cactvs(osra_program)*
  The name or full path of a local installation of the **Osra** chemical **OCR** program. If it is found, it is used in preference to the Web service, which is advantageous in case of confidential data.
- *cactvs(overwrite_protection)*
  If this flag is set, most toolkit chemistry commands which write files (but not standard **Tcl** or **Python** file commands) will refuse to overwrite existing files, even if the file permissions would allow it

  This is a thread-local flag. Changing it has an effect for future threads only if the change is performed in the base thread. Otherwise, new threads inherit the current setting of the base thread.
- *cactvs(paper_size)*
  The default paper size. Usually set to *A4*, or *Letter*, but the standard range of DIN and US sizes is recognized.
- *cactvs(persistent_bond_set)*
  Bit set of bonds which are persistent, meaning that they are not completely discarded when the structure changes. The default set consists of the bond types *normal, dative, ionic, complex, 3center*, *nobond*, and *rgroup*.
- *cactvs(platform)*
  Platform information - *unix, mac* or *windows*.
- *cactvs(preload_datatype_extensions)*
  A list of datatype handler extension modules which should be automatically loaded at start-up (as per `typex load` command). This variable is typically set via a start-up configuration file since modifying it at run-time has no effect.
- *cactvs(preload_db_extensions)*
  A list of database connector extension modules which should be automatically loaded at start-up (as per `dbx load` command). This variable is typically set via a start-up configuration file since modifying it at run-time has no effect.
- *cactvs(preload_file_extensions)*
  A list of structure and reaction I/O extension modules which should be automatically loaded at start-up (as per `filex load` command). This variable is typically set via a start-up configuration file since modifying it at run-time has no effect.
- *cactvs(preload_network_extensions)*
  A list of network I/O extension modules which should be automatically loaded at start-up (as per `netx load` command). This variable is typically set via a start-up configuration file since modifying it at run-time has no effect.

- *cactvs(preload_table_extensions)*
  A list of table I/O extension modules which should be automatically loaded at start-up (as per **`tablex load`** command). This variable is typically set via a start-up configuration file since modifying it at run-time has no effect.
- *cactvs(processor_count)*
  The number of processor cores seen by the interpreter. There is no distinction between multi-core and multi-die processors. This attribute is read-only.
- *cactvs(property_definition_count)*
  The number of properties currently registered in the core. This attribute is read-only.
- *cactvs(property_lock)*
  If set, properties cannot be deleted from the internal property database, and many operations changing the character of a property are also disabled. This is normally only used in multi-threaded scripts where locking the property database can give a significant speed boost because it eliminates the need to look and unlock individual property definitions.
- *cactvs(propertypath)*
  The search path for property definitions.
- *cactvs(proxy_bypass_hosts)*
  A comma-separated list of host names which should be contacted directly, not via the proxy host. This variable is only used if a proxy has been configured. The default is an empty list.
- *cactvs(proxy_host)*
  The name of a proxy server which is used as an indirect Internet access gateway. All standard Internet-access operations will automatically reroute requests via this proxy, if a proxy host is set. If the string is empty, which is the default, direct Internet access without a proxy is used.
- *cactvs(proxy_password)*
  A password which is supplied to the Internet proxy server, if one has been configured.
- *cactvs(proxy_user)*
  The username to supply to the Internet proxy server, if one has been configured.
- *cactvs(pubchemhost)*
  The host name which is used for PubChem-related data retrieval via the **PUG** interface, or screen scraping of certain information pages. The default is *pubchem.ncbi.nlm.nih.gov.*
- *cactvs(python_error_nessage)*
  The last error message generated by a **Python** interpreter. If the current application does not contain an embedded **Python** interpreter, this is always an empty string. This attribute is thread-specific (though currently there can only be a single **Python** thread).
- *cactvs(python_object_autodelete)*
  This flag controls whether in the **Python** interface major objects behave like in **Tcl**, where they are persistent until explicitly deleted, or in standard **Python** fashion. In the latter style, objects are deleted when their last **Python** reference goes out of scope and it is not otherwise protected from deletion (such as being property data, or a member of a dataset). Objects which never instantiate a **Python** reference are never auto-deleted either. For compatibility, the default mode is the **Tcl** style. In this mode, major chemistry objects are only deleted by

explicitly calling their `delete()` methods, or when used in a `with` clause. The `del` command only deletes (more precisely: decrements the reference count of) the associated **Python** wrapper object, without touching the underlying chemistry object, and variable scope tear-downs have no effect either. If auto-delete is enabled, the deletion of reference objects by the latter two methods also deletes unprotected chemistry objects. `delete()` and `with` clauses continue to work as in the **Tcl** mode.

- *cactvs(python_support)*
  A boolean read-only flag indicating whether a parallel **Python** interpreter is available in this application, which can be accessed with the `python` command, or indirectly by using properties with computation functions written in **Python**.
- *cactvs(qlz_support)*
  A read-only boolean read-only attribute indicating whether the object swap system uses the faster **QLZ** compressor instead of slower **ZLIB** compression.
- *cactvs(reaction_count)*
  The current number of reaction objects. Note that not all reactions necessarily have **Tcl** handles or **Python** wrapper objects, so there may be a discrepancy between this value and the result list length of the `reaction list` command.
- *cactvs(release_date)*
  Release date of the toolkit, in standard Unix format as seconds since 1970, intended for use with the `clock format` command. Read only.
- *cactvs(repository_support)*
  A read-only boolean flag indicating whether this interpreter supports repository objects.
- *cactvs(repxpath)*
  The alternate representation module search path
- *cactvs(ring_bond_set)*
  Bit set of bond types which define rings in structures. The default set consists of the bond types *normal, dative, ionic, complex*, and *3center.*
- *cactvs(ringset)*
  The type of ring set computed by default. The default is *esssr.*
- *cactvs(rpc_id)*
  ID of the **RPC** communication protocol between **Cactvs** processes.
- *cactvs(rpc_support)*
  A read-only boolean flag indicating whether this application was compiled with support for **RPC**-based socket-facilitated data exchange between **Cactvs** applications, for example remote property calculations. This flag only informs about that specific data exchange mechanism. The availability of other socket-based mechanisms (remote datasets, remote query processing, table row data transfer, file access via **URL**s, etc.) are not covered.
- *cactvs(rpc_version)*
  The version of toolkit-internal **RPC** communication protocol.

- *cactvs(safe_interpreter)*
  A boolean read-only flag. If set, the current interpreter is safe, i.e. has been stripped of potentially dangerous commands.
- *cactvs(script)*
  The read-only name of the currently executed application script. Empty if no script is run. Depending on the main interpreter type, this is either a **TCL** or a **PYTHON** script.
- *cactvs(script_url)*
  If the script interpreter executes a script obtained by Internet access via an **URL**, the **URL** is listed here. Read only.
- *cactvs(secondary_script)*
  The read-only name of the secondary application script, i.e. the helper **PYTHON** script when running a main **TCL** script, or the helper **TCL** script when running a main **PYTHON** interpreter. If no such string is used, this is an empty value.
- *cactvs(setsize_exceeded)*
  A thread-local flag indicating that a command which allows the specification of a maximum number of result objects to create has attempted to exceeded this number. Commands which can be controlled in this manner reset the variable to 0 when the command execution begins. Example: The **`ens transform`** command.
- *cactvs(smartsmacro_table)*
  The handle of the table with **SMARTS** macro expansion data (usually this is *table2*). This attribute is read-only.
- *cactvs(smiles_hypervalent_hydrogen_addition)*
  If this flag is set, the addition of implicit hydrogens follows the strict Daylight specification demanding that hydrogens are added until the next standard valence is reached. This means that, for example, an uncharged S(III) or S(V) atom is hydrogenated to S(IV) or S(VI). Chemically, this usually does not make sense, but it can be useful for interoperability testing or porting of legacy Daylight functionality. By default, hydrogens are not added to hypervalent atoms in PSE groups 15 and 16, which makes automatic correction of the structure to much more likely S^+(III) and similar species simpler.
- *cactvs(smiles_version)*
  The version of `SMILES/SMARTS` to use. Currently, it can be set to 4.9 or 4.3 (or any value below 4.9). The difference is that in version 4.9 the *x* atom attribute is interpreted according to the newly introduced **DAYLIGHT** definition (number of ring bonds), while before that this letter was unused and interpreted as an **CACTVS** extension (number of hetero atom ligands). In 4.9 mode, the former *x* extension is now accessed via the letter *z*. The default is 4.9, i.e. full support of the most current **DAYLIGHT** definition.
- *cactvs(smtp_host)*
  The default mail host to use when sending email with the **`mail`** command.
- *cactvs(smtp_password)*
  The password to use when connecting to above mail host.

- *cactvs(smtp_port)*
  The port to use to connect to the mail host. The default is 587, the newer recommended value both for *smtp* and *smpts* transfers. Some mail host may require other ports, such as the old standard 25, or the alternative port 465 for direct *smpts* without negotiation.
- *cactvs(smtp_user)*
  The user ID to use when connecting to above mail host.
- *cactvs(soap_method)*
  Name of default method called via **SOAP** request.
- *cactvs(soap_schema)*
  Default **SOAP** communication schema name.
- *cactvs(soap_uri)*
  Default **URI** associated with **SOAP** communication schema.
- *cactvs(source_lookup)*
  If this boolean flag is set, the toolkit will attempt to retrieve source code of modules obtained via Internet connections, in addition to the executable format.
- *cactvs(sourcepath)*
  The search path for source code for extension modules.
- *cactvs(sql_dialect)*
  The **SQL** dialect to be used when generating **SQL** code fragments, such as in a `sqlget` command. This can be the name of any database connector module, though currently dialect-specific formatting is only used for *mysql*, *postgresql* and *sqlite*.
- *cactvs(ssl_client_cert_file)*
  The name of the **SSL** client certificate file if one was specified at start-up.
- *cactvs(ssl_server_cert_file)*
  The name of the **SSL** server certificate file if one was specified at start-up. The certificate is only used for acting as a **KNIME** node processing server with encrypted communication support. Fetching data from encrypting Internet sites, such as via *https*, does not require a server certificate.
- *cactvs(ssl_server_key_file)*
  The name of the **SSL** server key file if one was specified at start-up. The certificate is only used for acting as a **KNIME** node processing server with encrypted communication support. Fetching data from encrypting Internet sites, such as via *https*, does not require a server certificate.
- *cactvs(ssl_server_support)*
  A read-only flag indicating if the software was compiled with support for acting as an **SSL**-enabled **KNIME** node processing server, and additionally a valid certificate and key file set was provided at start-up and successfully used for initial **SSL** configuration. If no cert file is specified explicitly, a default *localhost* certificate/key file set stored in the *knime* subdirectory of *::cactvs(data_directory)* is used, if the application was compiled with **SSL** server support and the file can be found.

- *cactvs(standalone)*
  A read-only boolean flag which is set for applications which encapsulate an application script and an interpreter into a single, stand-alone executable.

- *cactvs(stationpath)*
  The directory path where to look for auto-loaded station definition files.

- *cactvs(strict_kekulization)*
  If this flag is set, failures to generate a complete Kekulé structure when reading formats which encode aromatic bonds without an explicit bond order (such as some **SMILES** schemes) raises an error. By default, a structure with a partially resolved Kekulé system or single bonds is generated if no full resolution can be found, which might still be useful for structure matching, feature inspection, etc.

- *cactvs(strict_smarts)*
  If set, **SMARTS** specifications are interpreted according to strict Daylight specification. The default interpretation is more lenient, for example the aliphatic attribute of upper-case element symbols is not enforced to avoid the need for lengthy aromatic/aliphatic alternative symbols or # notation. Even without this flag, there are various methods to decode a substructure specification according to strict Daylight rules, for example by appending a ! character to the **SMARTS** string, or using appropriate decoder flags in **`ens create`** or **`molfile read`** commands. This option only applies to newly decoded **SMARTS** substructures. Substructures read from files such as *cbin/cbase/bdb* which encode the internal object structure of the toolkit already contain the interpretation implicitly and persistently as part of the atom and bond attribute set.

- *cactvs(structure_security)*
  If this boolean flag is set, no structure or reaction information is sent over Internet connections as a decodable connection table. It is off by default for academic distributions, but on for commercial packages.

  If it is on, Internet lookup is still allowed by means of sending identifiers which are likely to be non-confidential (for example, **CAS** numbers, or **PubChem** CIDs and SIDs) or which cannot be reversed (**Cactvs** hash codes). However, implicit sending of the structure as connection table to Internet services which provide computation or identifier lookup is suppressed if the flag is on. For example, computation of property `E_CID` for the PubChem identifier by means of contacting the **PubChem** database then always fails, regardless whether the structure has a CID or not, because that operation would require a potentially confidential structure to be sent as decodable InChI string to that database.

  This flag can be changed by developers who know what they are doing. Also, direct scripting of any type of Internet access is possible regardless of the flag setting. This flag is a basic safeguard against unintentional data leakage.

  The attribute is thread-local. Changing it has sets the default for future threads only if the modification is performed in the base thread, otherwise these inherit the current setting in the base thread.

- *cactvs(superatom_table)*
  The handle of the table with superatom expansion data (usually this is *table1*). This attribute is read-only.

- *cactvs(swap_count)*
  The number of objects which are currently swapped out (see `ens swapout`, `reaction swapout` commands). This element is read-only.
- *cactvs(swap_directory)*
  The name of a directory to use for ensemble and reaction swapping. By default it is initially an empty string, which is replaced by an automatically created process-specific subdirectory of the system temp directory once the swap subsystem has been initialized.

  If this element is set to the name of an existing, writable directory before the first swap operation commences, that directory will be used instead. Changing the control variable value after the first object has been swapped has no effect. An automatically set up swap directory is deleted upon program shutdown. In a user swap directory, only the swap files are removed when the program exits.
- *cactvs(swap_store_type)*
  Select the type of object store used to swap out excessive numbers of ensemble and reaction objects (see for example the *swapthreshold* `dataset` object attribute). The default store is *gdbm* (simple key/value store), but it may also be set to *none* (use separate native **CACTVS** binary files) or *tokyocabinet* (advanced key/value store). The *tokyocabinet* store is not supported on Windows. Setting this attribute has only an effect before the first object is swapped out. If a method is set which is not supported, the best available method is instead automatically selected when the first object is swapped out, and the array variable element updated to the actually employed method.
- *cactvs(swap_support)*
  A read-only boolean flag indicating whether this interpreter supports the swapping of objects to a disk store.
- *cactvs(system_encoding)*
  The currently configured system encoding. By default, it is the same as `::cactvs(default_system_encoding).`
- *cactvs(table_count)*
  The current number of table objects in the application. Note that not all tables necessarily have **TCL** handles or **PYTHON** wrapper objects, so there may be a discrepancy between this value and the result list length of the `table list` command.
- *cactvs(table_support)*
  A read-only boolean flag indicating whether this application was compiled with support for the chemistry-aware table object, and the associated `table` and `tablex` commands.
- *cactvs(table_swap_threshold)*
  Maximum object count of the datasets embedded in tables before disk swapping of these objects is automatically initiated. The default value is 10000. A negative value disables automatic disk swapping. This object-specific control value overrides the generic `dataset` object swap threshold in `::cactvs(dataset_swap_threshdold).`
- *cactvs(tablexpath)*
  The search path for table I/O module extensions

- *cactvs(tcl_error_message)*
  The last **Tcl** error message generated by the interpreter. This is a read-only element, and thread-local.
- *cactvs(tcl_reference_objects)*
  A flag indicating what type of **Tcl** interpreter data value objects are used for toolkit objects. The default value of zero indicates that object handles and object labels returned by executed commands are simple strings or numbers which are decoded when a new command is executed which uses them as arguments. A value of one indicates that the interpreter objects are actually wrappers for pre-resolved pointers. Using reference objects speeds up scripts, but slightly changes the pattern of handle allocations, and under many circumstances requires extra memory because toolkit objects remain in a zombie state until all interpreter references to a chemistry object have been reclaimed.
- *cactvs(tcl_support)*
  A read-only boolean flag indicating whether this interpreter support **Tcl** as scripting language. If seen from the **Tcl** side, this is obviously always true, but there may be **Python**-only interpreters where this flag is not set.
- *cactvs(tcl_thread_count)*
  A read-only integer with the total number of **Tcl** main interpreter threads. It is incremented for example by using `thread::create` or `dataset addthread` commands.
- *cactvs(test_interpreter)*
  The name of a **Tcl** slave interpreter which has been set up for regression testing. Usually an empty string and not used in user applications.
- *cactvs(testdatapath)*
  The search path for test data associated with extension modules for self-tests.
- *cactvs(thread_count)*
  The current number of threads registered with the core library. Every `Tcl` thread is also a library thread, but not every thread needs to be `Tcl`-enabled.
- *cactvs(thread_support)*
  A read-only boolean flag indicating whether the core library supports multiple threads. This is not equivalent tot he *multithreaded* flag, which indicates that multi-threaded **Tcl** script interpreters are supported. There are configurations which do not allow multiple parallel scripts, but support implicit multi-threading on operations such as file scanning from a single Tcl interpreter.
- *cactvs(timezone)*
  The current time zone in minutes west of GMT. This is used when setting up and storing time/date values. If the system clock is on GMT time, the value is always zero.
- *cactvs(tk_support)*
  A read-only boolean flag indicating whether this interpreter supports the **Tk** graphical toolkit as a compiled-in module. If not set, it is still possible to load **Tk** as dynamic module with the standard `load` or `package` **Tcl** commands.

- *cactvs(tmpdir)*
  The name of the directory for temporary files. This path can be changed and all future temporary files are then placed in that directory - and also automatically deleted if their controlling object is deleted. By default its value is the standard system temp directory.

- *cactvs(tokyocabinet_support)*
  A read-only boolean attribute indicating support of the Tokyo Cabinet key/value store. It this is available, the default object swap store is a Tokyo Cabinet file (instead of *gdbm* or separate native binary object serialization files), and the `tc` **TCL** command extension module is present. Currently, the Windows version of the **CACTVS** suite does not support Tokyo Cabinet.

- *cactvs(trace)*
  A bit combination of flags to trace various subsystems. Symbolic names for the subsystems are translated into the proper bit. Example:

  ```
  set cactvs(trace) ss
  ```

  traces substructure matching.

- *cactvs(tre_support)*
  A read-only boolean flag indicating whether this interpreter supports extended regular expressions and approximate regular expressions by means of the **TRE** library, in addition to standard regular expressions. If it is not set, only standard **PCRE/PERL** regular expressions are supported.

- *cactvs(trusted_executable_directory)*
  The name of a directory which contains only trusted executables which may be invoked by the toolkit. Usually, this is the installation directory for executables bundled with the toolkit.

- *cactvs(trusted_objectpath)*
  A search path for trusted shared libraries, OSX bundles and Windows DLLs implementing extension modules. Depending on the extension loaded mode, objects in this path may still be loaded even if the loading of modules from the full set of search paths is disabled. See also `::cactvs(dynamic_loaded_mode)`.

- *cactvs(trusted_scriptpath)*
  A search path for sources of trusted scripts. Scripts from this source may be loaded and executed even if normal script auto-loading is disabled.

- *cactvs(typexpath)*
  The search path for handlers of additional data types.

- *cactvs(update_package)*
  The package name this software was installed under, and which is used as file name component to find suitable updates in combination with the version and operating system data. For standard packages, this is *cactvstools*.

- *cactvs(update_url)*
  The directory **URL** where updates for this installation can be found. For commercial customers, this leads to a specific private area on our server, the academic version page is at *https://xemistry.com/academic*.

- *cactvs(use_bad_property_cache)*
  If this boolean flag is set, the toolkit remembers property names for which the look-up of property definition records failed. Since the look-up path can potgentially involve lengthy database searches and Internet accesses, this flag can in some cases prevent poor performance because of repeated failure to identify the same property name. On the other hand, if this flag is set, adding a new property definition during the runtime of an application after a look-up for this new property has failed has no effect and such a property definition needs to be loaded explicitly via its file.
- *cactvs(user_address_city)*
  The city part of the user contact address.
- *cactvs(user_address_country)*
  The country part of the user contact address. The country name must follow **ISO3166**.
- *cactvs(user_address_state)*
  The state part of the user contact address. Empty if no applicable.
- *cactvs(user_address_street)*
  The street address of the user contact address. Includes floor, house number, etc.
- *cactvs(user_address_zip)*
  The **ZIP** code (or equivalent postal processing code) of the user address.
- *cactvs(user_affiliation)*
  The company, university or other institution the user is working for. This information is used in pre-setting various header data blocks. The veracity of this item is not checked. It is typically set by a configuration file.
- *cactvs(user_affiliation_duns)*
  The **DUNS** number of the user institution.
- *cactvs(user_affiliation_url)*
  A **URL** describing the user institution.
- *cactvs(user_email)*
  The e-mail address of the user, initially guessed by the library. This information is used in pre-setting various header data blocks, and also for Web data sources which require some identification. The veracity of this item is not checked. It is typically set by a configuration file.
- *cactvs(user_id)*
  The current system user name, as extracted from the login data or similar sources. Read only.
- *cactvs(user_name)*
  The full name of the user, if it could be determined from password database information. This information is used in pre-setting various header data blocks. The veracity of this item is not checked. It is typically set by a configuration file.
- *cactvs(user_orcid)*
  The **ORCID** of the user. The veracity of this item is not checked. It is typically set by a configuration file.

- *cactvs(user_phone)*
  A phone number of the user.
- *cactvs(user_url)*
  A **URL** with info on the user. The veracity of this item is not checked. It is typically set by a configuration file.
- *cactvs(uuid_support)*
  A read-only flag indicating whether the toolkit was compiled with support for handling and generating **UUID**s. This is usually true.
- *cactvs(value_buffer_size)*
  The maximum size of a single property data element may request when it is converted to a string. Normally set to something big like 256K, but can be reduced to cope with external tools which have maximum line sizes, etc.
- *cactvs(verify_modified_property_values)*
  If set, property values which are set or modified by script commands are immediately verified and the command fails of the verification fails. The verification process checks the new value complies with any constraints defined as part of the property definition, and whether the property check function (if it exists) also accepts the new value.
- *cactvs(version)*
  Version number of the **CACTVS** toolkit. Read only.
- *cactvs(wedge_interpretation)*
  This attribute can be used to switch the stereochemical interpretation of wedge bonds. The default model is *cactvs*, the other possible value is *idbs* for compatibility with **IDBS** software. In **CACTVS** mode, the stereo center is always at the wedge tips. In **IDBS** mode, bonds which go down have the wedge base at the stereo center. This flag also changes the style of generated wedges in 2D layouts (properties `A_XY` and `B_FLAGS`).
- *cactvs(wrapper)*
  A read-only string which holds the name of a wrapper program (usually a Bourne shell or **MSDOS** script) invoking the interpreter, and possibly also setting a program script as hidden argument. For standard **CACTVS** installations, these are the *cs??* scripts. In case a raw interpreter without a wrapper is run, this string is empty.
- *cactvs(xdr_support)*
  A read-only boolean attribute which indicates whether this application supports **XDR**-based **CACTVS** object streams. This feature is required for native **CACTVS** file I/O, handling serialized packed object strings, the object swap store, and other toolkit functions.
- *cactvs(xml_support)*
  A read-only boolean attributes which indicates whether the application was compiled with an integrated **XML** parser. This affects the availability of the `soap` command, as well as support for various **XML**-based chemical structure and reaction or table data file formats.

Standard Filters

This section lists the filters which are built into the standard library or are shipped as filter definition files in the standard distribution.

A more detailed description of the filters can be found in the auto-generated documentation file *$::cactvs(data_directory)/autodocs/filters.html*

The following table contains the filter name in the first column and the property the filter operates on in the second column. The third column is an explanation.

Standard Filter Table

0neighbors 1neightbor 2neighbors... 12neighbors	`A_NEIGHBORS`	An atom with a specific number of bonded neighbor atoms.
3datom	`A_TYPE`	An atom or pseudo atom which may have a 3D coordinate. This includes classical atoms, 3D points, open valences, electron pairs, super atoms and polymers.
3dplotatom	`A_XYZ`	Atom has defined 3D coordinates
3dplotbond	`A_XYZ`	All bond atoms have defined 3D coordinates
3dpoint	`A_TYPE`	An pseudo atom which is just a point in 3D space, but probably with property data such as NMR shielding values which are also applicable to classical atoms.
acidicatom	`A_HYDROGEN_BONDING`	A hydrogen atom which is acidic.
acspecial	`A_CSPECIAL`	Atom is a special carbon (i.e. is usually drawn with atom symbol)
ahspecial	`A_HSPECIAL`	A special hydrogen, i.e. one which is usually displayed in depictions, such as hydrogen at stereo centers or stereo double bonds, or on aldehydes.
ametal	`A_ISMETAL`	A metal atom.
anion	`A_FORMAL_CHARGE`	Atom has a negative charge.
anyatom	`A_QUERY`	The atom is an *any* atom specification for substructure searching.
aopen	`A_TYPE`	Open valence pseudo atom
apicenter	`A_PICENTER`	An atom which is a π center.
aradical	`A_RADICAL`	An atom with unpaired electrons.
aroatom	`A_ARORING_COUNT`	An atom which is part of an aromatic system.
arobond	`B_ARORING_COUNT`	A bond which is part of an aromatic system.
aroring	`R_AROMATIC`	An aromatic ring.
aroringsystem	`Y_AROMATIC`	A ring system that is fully aromatic, i.e. all rings in it are aromatic rings.
arotautobond	`B_ISAROTAUTOMERIC`	Test whether the bond is within an extended (possibly aromatic-system-spanning) tautomer system
arrowbond	`B_FLAGS`	The bond is marked with one of the display attribute bits for parallel arrows.

Standard Filter Table

asidechain	`A_FRAMEWORK`	An atom which is a side chain in the framework nomenclature (ring/chain/linker).
astereogenic	`A_STEREOGENIC`	An atom which can be a stereo center.
aunmapped	**`A_MAPPING`**	Atom which is unmapped
aunsub	`A_HETERO_COUNT`	An atoms which is not substituted by hetero atoms.
basicatom	`A_HYDROGEN_BONDING`	An atom which is basic, i.e. is easily protonated.
bondangle	`B_TYPE`	A filter for pseudo bonds of type *bond angle*. The toolkit supports as a pseudo-bond type the encoding of bond angles (angle between two bonds, involving three atoms with one atom common to both bonds).
bondtorsion	`B_TYPE`	A filter for pseudo bonds of type *torsion angle*. The toolkit supports as a pseudo-bond type the encoding of torsion angles (angle between the planes formed by three atoms each, with two atoms joined by a common bond in the middle).
boronatom	`A_ELEMENT`	The element of the atom is boron.
boxedatom	`A_FLAGS`	The boxed attribute has been set for the display flags of the atom.
bromineatom	`A_ELEMENT`	The element of the atom is bromine.
bstereogenic	`B_STEROGENIC`	The bond can be a stereo center.
carbocycle	`A_ELEMENT`	The ring contains only carbon.
carbonatom	`A_ELEMENT`	The element of the atom is carbon.
cation	`A_FORMAL_CHARGE`	Atom with positive formal charge
cbond	`A_ELEMENT`	A bond with one or more carbon atoms in it.
ccbond	`A_ELEMENT`	The bond is a carbon-carbon bond. This is a real bond filter - the atom property is used in a special mode to check on all bond atoms directly
chalcogen	`A_IUPAC_GROUP`	The atom is a chalcogen (**PSE** group VIa).
chargedatom	`A_FORMAL_CHARGE`	The atom bears formal charge.
chbond	`A_ELEMENT`	The bond is a carbon-hydrogen bond. This is a real bond filter - the atom property is used in a special mode to check on all bond atoms directly
chlorineatom	`A_ELEMENT`	The element of the atom is chlorine.
classicatom	`A_TYPE`	A classical atom from the periodic system. The toolkit supports operation with pseudo atoms, such as superatoms, atom query specifications, or 3D points. The class of atoms is set in the property `A_TYPE`. This filter checks that `A_TYPE` is *normal*
classicbond	`B_TYPE`	A classical bond between two classical atoms with a defined bond order and valence electrons which are subtracted from the free electron count of the participating atoms. The type of bonds is stored as data of property `B_TYPE`, which is also the property this filter operates on.
classicmol	`M_ATOM_RANGE`	The molecule/fragment has at least one classic atom.

Standard Filter Table

classicring	`A_TYPE`	The ring only consists of classical atoms and no pseudo atoms, such as 3D points, superatoms, query specifications, etc
cneighbor	`A_ELEMENT`	The atom is bonded to at least one carbon atom.
complexbond	`B_TYPE`	The bond is of type *complex*.
crossedbond	`B_TYPE`	The bond has the *crossed* display attribute set.
cxbond	`A_ELEMENT`	The bond is a carbon-hetero bond.
dashbond	`B_FLAGS`	The bond has any of the display attribute bits for dashed/stippled rendering set.
doublebond	`B_ORDER`	The bond is a double bond (bond order 2). This includes Kekulé bonds in aromatic systems.
edgebond	`B_FLAGS`	The bond is displayed. This bit is usually set by the **Tk** **`plotbond`** command and is not computable.
envelopering	`R_TYPE`	The ring is an envelope ring.
epairatom	`A_FREE_ELECTRONS`	Atom with two or more electrons not participating in VB bonds.
esssrring	`R_TYPE`	The ring is in the ESSSR set
flagbond	`B_FLAGS`	The bond has any display attribute flag set.
fluorineatom	`A_ELEMENT`	The element of the atom is fluorine.
h0atom *h1atom* *h2atom* *h3atom*	`A_HCOUNT`	The atom has exactly *n* bonded hydrogens. This are aliases for filter names *hydrogen0..3*.
hacceptoratom	`A_HYDROGEN_BONDING`	The atom is a hydrogen acceptors, i.e. its `A_HYDROGEN_BONDING` value is either *basic* or *acceptor.*
halogen	`A_IUPAC_GROUP`	The atom is a halogen (**PSE** group VIIa).
hdonoratom	`A_HYDROGEN_BONDING`	The atom is a hydrogen donor, i.e. its `A_HYDROGEN_BONDING` value of either *donor* or *acidic*. This attribute applies to the hydrogen atom directly, not the atom the hydrogen atom is bonded to.
hbond	`A_ELEMENT`	A bond with one or more hydrogen atoms participating.
heteroatom	`A_ELEMENT`	The atom is a hetero (not C, not H) atom.
heterocycle	`A_ELEMENT`	The ring is heterocycle.
highlightatom	`A_FLAGS`	An atom where the *highlight* flag in `A_FLAGS` is set
highlightbond	`B_FLAGS`	A bond where the *highlight* flag in `B_FLAGS` is set
hneighbor	`A_ELEMENT`	The atom is bonded to a hydrogen atom.
hxbond	`A_ELEMENT`	A bond between hydrogen and hetero atom.
hydrogenatom	`A_ELEMENT`	The atom is hydrogen.
interferingatom	`A_FLAGS`	Atom for which the interfering bit has been set, for example while interpreting a Lhasa transform
iodineatom	`A_ELEMENT`	The element of the atom is iodine.
ionicbond	`B_TYPE`	The type of the bond is *ionic*.
isotopeatom	`A_ISOTOPE`	The atom has an isotope label.

Standard Filter Table

kring	`R_TYPE`	The ring is in the K set.
largering	`R_SIZE`	The ring is larger than 7 members.
lhasapathatom	`A_LHASA_PATH_LABEL`	The atom is in the path, as matched by a pattern in a Lhasa transform
listatom	`A_QUERY`	The atom is an element list for substructure searches.
macrocycle	`R_SIZE`	The ring is of size 12 or more.
magent	`M_REACTION_ROLE`	Molecule has *agent* reaction role.
mbyproduct	`M_REACTION_ROLE`	Molecule has *byproduct* reaction role.
mcatalyst	`M_REACTION_ROLE`	Molecule has *catalyst* reaction role.
mediumring	`R_SIZE`	The ring is of size 5-7.
mimpurity	`M_REACTION_ROLE`	Molecule has *impurity* reaction role.
minorganuc	`M_ORGANIC`	The fragment is inorganic according to the normal definition.
mintermediate	`M_REACTION_ROLE`	Molecule has *intermediate* reaction role.
morganic	`M_ORGANIC`	The fragment is organic according to the normal definition
mproduct	`M_REACTION_ROLE`	Molecule has *product* reaction role.
mreagent	`M_REACTION_ROLE`	Molecule has *reagent* reaction role.
msolvent	`M_RACTION_ROLE`	Molecule has *solvent* reaction role.
multibond	`B_ORDER`	This is a multiple bond with bond order > 1. Kekulé aromatic bonds are included.
multiringatom	`A_RING_COUNT`	The atom is member in more than one (non-envelope) ring.
multiringbond	`B_RING_COUNT`	The bond is member in more than one ring.
multiringsystem	`Y_RING_COUNT`	The ring system consists of two or more rings.
mwaste	`M_REACTION_ROLE`	Molecule has *waste* reaction role.
nitrogenatom	`A_ELEMENT`	The element of the atom is nitrogen.
nodeatom	`A_FLAGS`	The atom is displayed as a node (bond line node or symbol) or is completely omitted (for example, normal hydrogen atoms). The `A_FLAGS` bit is usually set by the **Tk** **`plotatom`** command. However, it may also the set by a standard computation of `A_FLAGS`.
nofilter	`E_NONE`	This filter does nothing. Convenient in places where an empty string causes formatting or syntactic problems.
nomatchatom	`A_QUERY`	The atom is a substructure match specification which should *not* be matched.
openvalenceatom	`A_HYDROGENS_NEEDED`	The atom has open valences.
oxygenatom	`A_ELEMENT`	The element of the atom is oxygen.
participatingatom	`A_FLAGS`	Atom is participating in a reaction, outside of direct reaction center. This is for example annotated by Lhasa reaction transforms.
phosphorusatom	`A_ELEMENT`	The element of the atom is phosphorus.
plotatom	`A_XY`	The atom has valid 2D display coordinates.

Standard Filter Table

plotbond	`A_XY`	The bond has valid 2D display coordinates. This is indirectly determined via display coordinates of the atoms of the bond.
pnicogen	`A_IUPAC_GROUP`	The atom is a pnicogen (**PSE** group Va).
protectedatom	`A_FLAGS`	Atom which needs protection in a reaction. Annotated for example by Lhasa transforms.
queryatom	`A_TYPE`	An atom which is purely a query specification.
querytreeatom	`A_QUERY`	An atom with a query tree specification in `A_QUERY`.
querytreebond	`B_QUERY`	A bond with a query tree specification in `B_QUERY`.
racemicatom	`A_RACEMATE`	A stereogenic atom which is explicitly declared a racemate.
racemicbond	`B_RACEMATE`	A stereogenic bond which is explicitly declared a racemate.
reactionagent	`E_REACTION_ROLE`	The role of an ensemble in a reaction is *agent*.
reactionaltreagent	`E_REACTION_ROLE`	The role of an ensemble in a reaction is *altreagent*.
reactionbyproduct	`E_REACTION_ROLE`	The role of an ensemble in a reaction is *byproduct*.
reactioncatalyst	`E_REACTION_ROLE`	The role of an ensemble in a reaction is *catalyst*.
reactionimpurity	`E_REACTION_ROLE`	The role of an ensemble in a reaction is *impurity*.
reactionintermediate	`E_REACTION_ROLE`	The role of an ensemble in a reaction is *intermediate*.
reactionprecursor	`E_REACTION_ROLE`	The role of an ensemble in a reaction is *precursor*.
reactionproduct	`E_REACTION_ROLE`	The role of an ensemble in a reaction is *product*.
reactionreagent	`E_REACTION_ROLE`	The role of an ensemble in a reaction is *reagent*.
reactionsolvent	`E_REACTION_ROLE`	The role of an ensemble in a reaction is *solvent*.
reactionwaste	`E_REACTION_ROLE`	The role of an ensemble in a reaction is *waste*.
redatom	`A_COLOR`	The atom color is *red*.
repeatatom	`A_QUERY`	An atom which has a repeat count range for structure searches
rgroupanchor	`A_QUERY`	The atom is an R-group anchor.
rgroupatom	`A_QUERY`	The atom belongs to an R-group definition.
rgroupbond	`B_TYPE`	The bond is linking an R-group definition to a core structure.
ring3... *ring10*	`R_SIZE`	Ring is of the specified size.
ringatom	`A_RING_COUNT`	The atom is a a member of a ring.
ringbond	`B_RING_COUNT`	The bond is a a member of a ring.
ringlinkeratom	`A_FRAMEWORK`	An atom which is a ring system linker in the framework nomenclature (ring/chain/linker).
ringlinkerbond	`B_FRAMEWORK`	A bond which is a linker bond in the framework nomenclature (ring/chain/linker).
satring	`R_UNSATURATED`	The ring is saturated.
selectedatom	`A_FLAGS`	An atom where the *selected* flag in `A_FLAGS` is set
selectedbond	`B_FLAGS`	A bond where the *selected* flag in `B_FLAGS` is set
siliconatom	`A_ELEMENT`	The element of the atom is silicon.

Standard Filter Table

simplering	`R_TYPE`	The ring is a simple (non-envelope) ring
singlebond	`B_ORDER`	A single bond. This includes Kekulé bonds encode with a single bond order.
slashbond	`B_FLAGS`	The *slashed* display attribute bit has been set on the bond.
smallring	`R_SIZE`	The ring is of size 3-4.
spatom	`A_HYBRIDIZATION`	The atom is *sp*-hybridized.
sp2atom	`A_HYBRIDIZATION`	The atom is *sp2*-hybridized.
sp3atom	`A_HYBRIDIZATION`	The atom is *sp3*-hybridized.
sp3sp3bond	`A_HYBRIDIZATION`	The bond is between two *sp3*-hybridized atoms.
spiroatom	`A_SPIRO`	The atom is a *spiro* atom.
splitbond	`B_FLAGS`	The *split* display attribute bit has been set on the bond.
sssrring	`R_TYPE`	Ring is in the SSSR set.
standardatom	`A_TYPE`	An atom which is either a classical atom, or a search specification.
starredatom	`A_FLAGS`	The attribute bit for an asterisk marker is set.
stereoatom	`A_STEREOINFO`	The atom has a valid stereo descriptor.
stereobond	`B_STEREOINFO`	The bond has a valid stereo descriptor.
stereosphere	`A_STEREOGENIC`	An atom which is a ligand to a possible atomic stereo center
structurebond	`B_TYPE`	A bond or type *normal* or *complex*. These are the bonds which usually define which atoms in an ensemble belong to a common molecule.
sulphuratom	`A_ELEMENT`	The element of the atom is sulphur.
superatom	`A_TYPE`	The atom is a super atom pseudo atom.
superatomsgroup	`G_TYPE`	The group encodes a superatom.
symbolatom	`A_FLAGS`	The atom is displayed with a symbol, and not suppressed or only display as node without a symbol. The `A_FLAGS` bit is usually set by the **Tk `plotatom`** command. However, it may also be set by a standard computation of `A_FLAGS`.
tautoatom	`A_TAUTOMER_SYSTEM`	Atom is part of a tautomer system.
tautobond	`B_ISTAUTOMERIC`	The bond is part of a tautomer system.
terminalatom	`A_TERMINAL_DISTANCE`	The atom is terminal, i.e. among the outermost atoms in the structure.
terminalbond	`A_TERMINAL_DISTANCE`	The bond is terminal, i.e. leads to an atom which is among the outermost atoms in the structure.
triplebond	`B_ORDER`	A triple bond with bond order 3.
unsatring	`R_UNSATURATED`	The ring is unsaturated. Aromatic rings are not considered to be unsaturated. The *xunsatring* filter does not have this extra condition. *insatring* is an alias.
v0atom.. *v8atom*	`A_VALENCE`	An atom with specific valence. These filters may also be abbreviated as *v0, v1, .. v8.*
varbond	`B_QUERY`	The bond has been specified to match different bond orders in a substructure query specification.
wavybond	`B_FLAGS`	The wavy display attribute flag has been set for the bond.

Standard Filter Table

wedgebond	`B_FLAGS`	The bond has a wedge bit set
xbond	`A_ELEMENT`	A bond with one or more hetero atoms participating.
xneighbor	`A_ELEMENT`	The atom is bonded to hetero atom.
xunsatring	`B_ORDER`	This is a ring with double bonds in the ring. In contrast to the *insatring* filter, this filter does not take aromaticity into account. A phenyl ring with Kekulé double bonds passes this filter, but not the *insatring* filter.
xxbond	`A_ELEMENT`	A bond between two hetero atoms.

This table only lists the most important element filters (the built-in set, plus the standard set of external filter definitions). The full set of elements in the periodic system is also provided as an element filter. Initially, only the most common elements are entered as pre-defined filters in the filter table and are listed here. However, the name of any element in the **PSE** can be used as filter name, and the filter is then automatically created if it has not yet been set up. All simple element filters have an alias name which corresponds to their atomic symbol, so using filter *c* is equivalent to using the filter *carbon* or *carbonatom*.

Most of these filters have additional aliases, and some scripts still use the old, non-systematic filter names. In order to obtain the listed name of a filter from any alias, use the following command:

```
lindex [filter get $name aliases] 0
```

System Tables

CACTVS TCL interpreters with table support start up with three predefined tables, which are accessible via their standard handles *table0*, *table1* and *table2*. They are write-protected by default, but by setting the *editable* attribute this can be changed.

If an interpreter is started in a **KNIME** context, additional tables may be automatically present.

The PSE table

This table has handle *table0*. It contains basic physical and display data for all elements. The rows are addressable via the periodic system number or the standard element symbol. Row zero contains pseudo data for an undefined element.

The columns are:

- *symbol*
  The element symbol as a string.
- *name*
  The English name of the element.
- *number*
  The element number in the **PSE** as an integer.
- *group*
  The PSE group using new-style **IUPAC** group indices 1..18, plus 19 for lanthanides and 20 for actinides.
- *pserow*
  PSE row, starting with 1.
- shellelectrons
  The standard number of shell electrons.
- *ismetal*
  Flag for metals
- *weight*
  The standard atomic weight in gr/mol.
- *vdwradius*
  Standard Van-der-Waals radius in Ångstrom.
- *covradius*
  Standard covalent bonding radius in Ångstrom.
- *color1*
  Default rendering color for bright backgrounds
- *color2*
  Default rendering color for dark backgrounds.
- *valences*
  Standard accessible atomic valences in compounds with this element. This column is an integer vector. For example, for sulfur the cell contains the values 2, 4 and 6.

- *isotopes*
  Standard isotopes of the element occurring naturally, or with precedence for chemical use as isotope label etc. This column is a compound vector. Every vector element is a list of the nucleon count (integer), the isotope mass (double, gr/mol), the natural abundance (double, range 0..1) and the half life in seconds. Stable isotopes have an infinite half life.
- *allisotopes*
  As above, but with a full set of known isotopes, obtained from *http://www.sisweb.com/referenc/source/exactmas.htm*
- *sequence*
  The precedence of the element in constructing an elemental formula, following the C/H/alphabetic convention (Hill formula). The value for C is one.

The Superatom table

This table has handle *table1*. It contains information about common superatom fragments. This information is for example used superatom expansion or contraction of chemical groups for display.

The columns are:

- *name*
  An expanded, human-readable name of the fragment
- *codes*
  A string list of recognized names for the fragment. There is often more than one code, and the list contains reversed entries such as *MeO* plus *OMe*, since superatom name matching does not automatically match reversed names - that could lead to mis-classifications in some cases.
- *smiles*
  A **SMILES** representation of the fragment. The first atom is always a * placeholder indicating the primary connection point. Secondary connection points may be present in any position. On expansion, the connection points are not used.
- *expandable*
  Flag indicating that this fragment is handled by normal expansion commands, such as **`ens expand`**.
- *contractable*
  Flag indicating that this fragment can be contracted as superatom in structure renderings.
- *level*
  The minimum contraction level to actually perform group contraction for rendering. This is currently a value between 1 (routinely abbreviated) to 3 (exotic case only recognized by specialists).
- *flags*
  Substructure match flags to be added to the standard set when trying to detect the fragment in a structure, such as forcing a match which does not link via a simple alkyl chain, preventing for example the use of an *n-propyl* group as a partial abbreviation for an *n-octyl* group.
- *textlabel*
  The standard display symbol of the contracted group, as it is stored in property `A_TEXTLABEL`.

- *structure*
  The handle of the decoded **SMILES** from the third column. Decoding is performed in a lazy fashion, so initially this cell is empty.
- *substructure*
  The handle of the **SMARTS**-decoded **SMILES** from the third column. Decoding is performed in a lazy fashion, to initially this cell is empty.

The SMARTS Macro table

This table has handle *table2* and is initially empty. If content is added, it is automatically used in all **SMILES** and **SMARTS** expansions which use macro syntax.

The columns are

- *name*
  A name for the macro. This is what is recognized in the input **SMILES**.
- *pattern*
  The **SMILES**/**SMARTS** pattern this macro expands to
- *set*
  A set name for this fragment. Set names are an argument to **SMILES**/**SMARTS** decoder functions. Only those macros in the table that match the set name are used, if it is not an empty string.

SMILES and SMARTS dialects

The toolkit supports the complete range of the Daylight **SMILES**, **SMARTS**, Reaction **SMILES** and **SMIRKS** standards, including Recursive **SMARTS**.

The global control variable `::cactvs(smiles_version)` can be set to a Daylight release number. The setting of this variable influences various aspects of encoding and decoding of **SMARTS** data. The default value is 4.9 - the version best known for finally introducing the *x* ring bond count atom attribute. This is the most recent major Daylight **SMILES**/**SMARTS** definition update.

In **SMARTS** context a simple 'H' atom attribute without a count is always interpreted by the toolkit as a hydrogen atom for explicit matching, not the hydrogen neighbour count. This behaviour is standard in Daylight tools since the 4.51 release.

Octahedral and bi-pyramidal stereochemistry in **SMILES** is read and written, but currently not checked by the substructure match routines. Allenes and square planar stereochemistry are fully supported.

Besides supporting the standard syntax and attributes of both atoms and bonds, a significant number of enhancements are also recognized:

Attribute ranges

In addition to a simple numerical count (as in '[X2]'), bracketed open and closed ranges are supported, as in '[X{1-}]', [X{-3}]' or '[X{2-3}]'. This feature is available for every attribute which can take a count. It is also possible to use the Eli Lilly operator extensions for the same purpose, as in '[X>1]' or [X<=3]'. The exception is the closed range, which cannot be expressed in Lilly syntax.

Match count prefixes

The **SMARTS** expression may be prefixed by a simple count, or an operator and a count. The **SMARTS** must then match the required number of times. The match mode is automatically adjusted if required. Example:

```
set ss [ens create {>4a-[F,Cl,Br,I]} smarts]
```

This matches compounds which contain 4 or more halogens substituting aromatic rings.

```
set ss [ens create {0[R]} smarts]
```

This matches compounds which do not contain rings.

Strict interpretation suffix

The default **SMARTS** interpretation in **CACTVS** is more lenient than the original Daylight definition. Specifically, the *aliphatic* attribute of upper-case element symbols is not enforced by default. Most match commands provide options to fine-tune the interpretation, and it is also possible to switch the toolkit globally into a strict **SMARTS** interpretation mode.

As a convenience, it is possible to request strict interpretation of a **SMARTS** string regardless of command options and global configuration by appending an exclamation mark to the string.

Example:

```
set ss [ens create C1CCCCC1!]
```

This **SMARTS** does not match benzene, which in default toolkit mode without the suffix is matched.

Additional atom attributes

- *a*
  Besides supporting its standard meaning without a suffix, the toolkit version allows a count to this attribute. If a count is set, the atom must be part of the count *or more* aromatic bonds. The associated property is `A_AROBOND_COUNT`. For example, **`[a3]`** matches the two central carbon atoms of naphthalene, but not the other ring atoms.

- *b*
  Followed by a 0 or 1 value, this attribute requires branching or chain character of the matched atom (i.e. up to 2, or 3 or more heavy atom substituents). The associated property is `A_SUBSTITUENT_COUNT`.

- *d*
  Heavy atom substituent count. Different from the *D* degree attribute, this one ignores even explicitly specified hydrogen atoms. A count suffix is required. The associated property is `A_SUBSTITUENT_COUNT`.

- *D*
  If used without a count, this symbol defines a deuterium atom.

- *e*
  An atom attribute for the ring pi electron count of all ESSSR rings the atom is part of. If there is more than one such ring, a match in any of these is sufficient. If no number modifier is supplied, the condition requires the presence of one or more pi electrons in the ring. The associated property is `R_PI_ELECTRON_COUNT`.

- *G*
  The same as the 'i' attribute. This is an Eli Lilly internal tools compatibility feature. Example:
  ```
  set ss [ens create {[aD3]-[G0;CH>0,O,N]} smarts]
  ```

- *HA*
  A hydrogen acceptor atom. This interpretation has precedence over the rather pointless *"hydrogen&aliphatic"* standard SMILES interpretation.

- *HD*
  A hydrogen donor atom.

- *i*
  An atom attribute checking for in/unsaturation. If a number modifier is specified, it requests a specific number of π bond participations (e.g. *i2* on carbon matches either an allene, or an alkyne). The associated property is `A_UNSATURATION`.

- *T*
  With a count, it is the same as the *z* attribute. This is an Eli Lilly internal tools compatibility feature. If used without a count, it defines a tritium atom. Example:
  ```
  set ss [ens create {[CT1]#C} smarts]
  ```

- *X*
  If used *without* a number modifier, which is illegal in standard **SMARTS**, this matches a hetero atom.

- *z*
  An atom attribute indicating a required number of hetero atom neighbors. If no numeric modifier is supplied, one or more hetero neighbors are required. The property associated with this attribute is `A_HETERO_SUBSTITUENT_COUNT`.
- *Z*
  An atom attribute indicating a required number of aliphatic hetero atom neighbors. If no numeric modifier is supplied, one or more hetero neighbors are required. The property associated with this attribute is `A_ALIHETERO_SUBSTITUENT_COUNT`.
- ^[0123456]
  An atom attribute where a following digit is required. This attribute checks the atom hybridization: 0=s, 1=sp, 2=sp2, 3=sp3, 4=sp3d, 5/6=sp3d2. The property associated with this attribute is `A_HYBRIDIZATION`.
- * and ?
  These two symbols both specify an 'any' atom.
- *#X*
  This atom symbol matches a hetero atom. It is a MOE compatibility feature.
- $$(...)
  This is a variation of normal recursive **SMARTS**. Standard recursive **SMARTS** does not know about atoms and bonds already matched in upper levels - the complete structure can be matched by the atoms in the recursive expression. This variant blocks all atoms and bonds already matched in any previous recursion level.
- |
  The vertical bar as bond symbol encodes a bond of type *complex*. This is a bond which is similar to a standard valence bond, for example with respect to defining molecular fragments, but is not electron-counted.
- */IWfss*
  An EliLilly extension: number of SSSR rings in the ring system the atom is a member of. Example:

  ```
  set ss [ens create {[/IWfss1o,s]1:c:c:c:c1} smarts]
  ```
- */IWspch*
  An EliLilly extension: The 0 or 1 suffix requires that the matched atom is part of the core, or the *Molecular Spinach* part of the structure. The associated property is `A_LILLY_SPINACH`, the literature reference is *J. Med. Chem. 2012, 55, 9763-9772*. Example:

  ```
  set ss [ens create {[/IWfss1o,s]1:c:c:c:c1} smarts]
  ```
- */IWhr*
  An EliLilly extension: Number of hetero atoms in one SSSR ring the atom is a member of. Example:

  ```
  set ss [ens create {[/IWhr1n]} smarts]
  ```
- */IWrid*
  An EliLilly extension. Atoms marked with the same ring ID must be a member of a common SSSR ring. Atoms with a different ring ID must be a member of at least one different SSSR ring. The set of chain atoms forms a pseudo-ring class and can also be tagged. Example:

```
set ss [ens create {Cl-[/IWrid1a].Cl-[/IWrid1a]} smarts]
set ss [ens create {Cl-[/IWrid1].Cl-[/IWrid2]} smarts]
```

- */IWfsid*
  As above, but applies to ring systems, not rings. Example:

  ```
  set ss [ens create {Cl-[/IWfsid1a].Cl-[/IWfsid1a]} smarts]
  ```

- */IWAr*
  An EliLilly extension. The attribute checks whether the atom is alpha to an aromatic ring. The associated property is `A_ALPHA_ARO_COUNT`. Example:

  ```
  echo [match ss {[/IWAr1Cl]} c1ccccc1Cl=CCl] (expect 1)
  ```

- */IWVy*
  An EliLilly extension. The attribute checks whether the atom is alpha to a vinyl, non-aromatic group. The associated property is `A_ALPHA_UNSAT_COUNT`. Example:

  ```
  echo [match ss {[/IWVy1Cl]} C=CCl] (expect 1)
  ```

Operator-chained matches

The toolkit supports to a limited degree the EliLilly extensions for chained matches. In these, multiple **SMARTS** fragments (which each may consist of multiple dot-disconnected parts) are linked via &&, || or ^^ two-character operators. Each fragment is handled independently, as a separate structure object, without regard to match overlaps as in Recursive SMARTS or explicit setting of the fragment overlap mode in substructure matching.

Example:

```
set ss [ens create {[nD3]-S(=O)(=O)&&0[aD3]-[G0;CH>0,O,N]} smarts]
```

The current implementation does not take operator precedence into account, as the original Lilly code does. It is possible to combine, for example, || and && parts in one query string, but the fragments are checked in strict left-to-right order, without precedence for the *and* part.

Only those parts of the expression are checked which are require to obtain the final match results. In case of an *or* expression, the match processing stops after the first fragment match has been found.

Eli Lilly extended SMARTS

As described above, the toolkit has near complete support for the published Eli Lilly SMARTS extensions, including match count prefixes, custom attributes, attribute count operators and chained matches.

Extended hydrogen handling

The H symbol may be used as an explicit hydrogen atom outside brackets, even though it is not in the official *organic subset* element set.

CACTVS Database Cartridges

The **CACTVS** toolkit is not just available as stand-alone script interpreter, or module which can be loaded into an existing **Tcl** or **Python** script interpreter, but also as a cartridge module for various databases.

The cartridge version extends the standard **SQL** function set of the core database by a number of additional functions (user-defined functions in **SQL** terms). The functions provide:

- Computation of chemical structure and reaction properties from decodable structure representations, such as **MDL SD** record, **SMILES** strings, or native **CACTVS** structure encodings. Typically these are taken from database table columns, but it is also possible to provide fixed values within a **SQL** statement.
- Perform structure query screening, match substructures or superstructures, or select similar structures in **SQL** queries. Various variants of full-structure matching are implicitly available by using standard **SQL** comparisons on columns containing structure hashes, or canonic strings such as InChIs.
- The capability to define **Tcl** processing scripts as custom functions. Both simple functions and aggregate functions can be defined. Within these functions, almost all of the toolkit functionalities can be used.
- Depending on the database system, additional helper functions, such as a formatter for bitvectors, may be available.

Supported Database Systems

The standard toolkit packages contain cartridges for **MySQL/MariaDB**, **PostgreSQL** and **SQLite**. Cartridge versions for **MS SQL Server**, **IBM DB2** and **Oracle** can be licensed separately.

The dependency of the cartridge on the exact database engine versions varies considerably. While there are multiple releases of the **PostgreSQL** cartridge for the major versions currently in use, there is only a single version of the **MySQL/MariaDB** and **SQLite** release, since these engines have rather stable internal data structures, unlike **PostgreSQL**.

Supported Functionality and Limitations

One important limitation of the cartridges is that they do not support the dynamic loading of extensions, neither by automatic mechanisms nor by explicit commands. This is similar to the secured *csweb* stand-alone interpreter for Web applications. Compared to standard basic toolkit interpreters, a lot of additional property definitions, I/O modules etc. are already compiled-in by default. Their availability can be queried by the normal script introspection commands. However, in case additional extensions are desired which are not contained in the default cartridge, a custom version of the cartridge must be compiled.

The cartridges are fully multi-threaded. There are no global locks which block parallel execution of multiple cartridge functions. Every script function instance in a **SQL** statement contains its own isolated, independent, and multi-threading script interpreter. The interpreters are automatically instantiated and taken down over the lifetime of a query. They do not interfere with other interpreters in parallel use, and they use the normal slave interpreter mechanisms to further encapsulate property computations performed within them. It is also possible to use another level of multi-threading within each of these interpreters. One may set up and use interpreter-global data which can be accessed over multiple function invocations of the same interpreter. This is mostly of relevance for aggregate functions. However, since the life of an interpreter ends when its associated **SQL** statement has been executed, it is not possible to set up persistent data accessible over multiple queries, except by using the file system, if that is allowed.

The cartridge utilizes an in-memory global chemistry object cache. This means, for example, that if there are multiple functions which use the same SD record from a table column as input, the file is in all likelihood (if there are no cache flushes) actually only decoded once and the obtained object re-used between functions, and even between multiple queries issued by the same or a different database user. Objects in the cache also retain their property data content, so if some property has already been computed for it, the information is returned directly, without re-computation. It is not possible to delete or edit cached chemistry objects. If such an operation is desired within a script, the object must be duplicated first, and then discarded when it is no longer needed.

Due to the severe multi-threading limitations of **Python**, the cartridges only support **Tcl** scripts.

Installing the Cactvs Database Cartridges

The exact procedure for cartridge installation depends on the database engine. The **SQL** installation script files, sample set-up scripts for the *ChEMBL* and *COD* databases (which can be freely downloaded as database dumps for local installation) and help texts as well as additional usage examples are located in the *sql* subdirectory of the central data directory of a standard toolkit installation.

MySQL/MariaDB Installation

- Copy the cartridge file to the **MySQL** plug-in directory, or create a link into that directory. This typically requires root permissions. The cartridge file is named *mysql_udfcactvsmodule.so* or similar, with the usual platform-dependent shared library suffix, and found in the *lib* subdirectory of a standard toolkit installation. The exact location of the **MySQL** plug-in directory is platform- and distro-dependent. A typical place is */usr/lib64/mysql/plugin*. The system documentation, as well as the contents of the */etc/my.cnf* and */etc/my.cnf.d/** files should be helpful to determine it. The **SQL** statement `show variables like 'plugin_dir'` can also be used to get the directory name.
- Next, find the file *mysqludf_init.sql* file in the *sql* subdirectory of the toolkit installation. Open it in a text editor and optionally perform the additional checks outlined in the first few comment lines. Then, log into the **MySQL** client console (the program name is usually simply *mysql*) with database administrator permissions and execute the command
  `source mysqludf_init.sql`
  The commands in that file make the functions on the cartridge file accessible to the SQL engine. If no error messages are printed, the installation is complete.
- In **MySQL**, **UDF** functions are defined globally. They directly become available in all databases served by the configured engine instance. The installation is also persistent and survives restarts of the database daemon.
- Applications which use the cartridge functions in **SQL** code they produce and transmit via the database client library do not need any special configuration, since the set-up is fully handled on the backend side.

PostgreSQL Installation

- Unfortunately, the internal data structures of **PostgreSQL** which are of importance to the cartridge are not stable between releases. The first step is therefore to determine the database engine version running on a system so that the right cartridge variant can be picked.
- Next, the **PostgreSQL** package directory needs to be found. Its actual location is highly system- and distro-dependent. Typical are directories like */usr/lib/postgresql94/lib64* or */usr/pgsql-9.6/lib*. It may be possible to retrieve this information via the command
  `pg_config --pkglibdir`
  if the database engine configuration query tools are installed on the system. Pick the suitable module library matching the **PostgreSQL** release from the provided file collection. The files are named something like *pg94_udfcactvsmodule.so*, *pg96_udfcactvsmodule.so*, *pg100_udfcactvsmodules.so* for the 9.4, 9.6 and 10.0 **PostgreSQL** releases. The right cartridge file must be copied or linked to the package directory, and **renamed** to the version-free generic name *pg_udfcactvsmodule.so*.

- Create the database you want the cartridge functions to use on if it does not yet exist. **UDF** definitions on **POSTGRESQL** are database-specific, not global. The functions need to be defined for each database, and re-defined if the database is deleted and re-created. Subsequent table creation or deletion within a database has no effect on the function accessibility.
- Find the function definition **SQL** script file *pgudf_init.sql* in the *sql* subdirectory of the normal toolkit installation. Read the first few comment lines and optionally perform the additional checks described therein. As a database administrator with suitable rights, run the command
  `psql $DBNAME -f pgudf_init.sql`
  to define the functions for the database. If no errors are reported, installation is complete.
- The installation is persistent on the database level and survives database daemon restarts. However, as described above, the function definition step (but not the shared library file copy operation) need to be repeated for each new or re-created database the cartridge functions should be accessible in.
- Applications which use the cartridge functions in **SQL** code they produce and transmit via the database client library do not need any special configuration, since the set-up is fully handled on the backend side.

SQLite Installation

- There is no persistent installation step for the **SQLITE** database engine. Every time the cartridge functions are to be used in a session, they must be re-loaded.
- Locate the *sqliteudf_init.sql* file in the *sql* subdirectory of the normal toolkit installation. Edit the path to the cartridge shared library file encoded therein. The run the command
  `.read sqlite_init.sql`
  on the *sqlite3* shell command line.
- In compiled applications which use the **SQLITE** library to access database files, you can add an appropriate call to the **`sqlite3_load_extension()`** function to automatize this process.

Property Computation with Cartridge Functions

There are four standard computation functions for structures/ensembles and four for reactions:

```
ens_string_property(structuredata,property,?parameters?,?format?)
ens_blob_property(structuredata,property,?parameters?,?format?)
ens_double_property()structuredata,property,?parameters?,?format?
ens_long_property(structuredata,property,?parameters?,?format?)
reaction_string_property(reactiondata,property,?parameters?,?format?)
reaction_blob_property(reactiondata,property,?parameters?,?format?)
reaction_double_property(reactiondata,property,?parameters?,?format?)
reacton_long_property(reactiondata,property,?parameters?,?format?)
```

Each of these has two mandatory and two optional parameters. The first parameter is the structure or reaction source specification. Often this is a column reference, but it can also be a constant string. All specifications which are accepted by the **ens create** and **reaction create** commands are possible. In case there could be ambiguity in the automatic format detection, the documented format prefixes are also recognized. If they are not present in the source, they could be added for example by a standard **SQL concat()** function:

```
select ens_double_property(concat("sln:",slncolumn),"E_WEIGHT");
```

The data types which are accepted by the structure or reaction data specification varies by database engine. Generally, these are any string or byte array/blob datatypes. While there is only a single instance for each of these functions for **MySQL**, which does not perform strict argument type checking and leaves the argument type checking to the function implementation, there are actually three versions of each function in **PostgreSQL**, accepting data types *text*, *varchar* and *bytea* as first argument.

The second argument is the name of the requested property in standard Cactvs notation. The retrieval of individual property fields is currently not supported. In normal applications, the argument is a constant string. Only properties where the associated object class corresponds to the function class are accepted. It is, for example, not possible to compute atom or reaction properties with an ensemble function. Only ensemble properties are accepted by the **ens_*()** functions. This is different from scripting with the standard toolkit property data retrieval commands.

Optional function arguments can simply be omitted, or a **NULL** constant passed.

The third, optional argument is a collection of parameters which should override the default parameters for the computation of the requested property. Its format is a key/value set in standard **Tcl** notation. The expected data type is a constant string. Example:

```
select ens_string_property("c1ncccc1","E_SMILES","usearo 0")
```

This command returns the encoding of the Pyridine input structure with Kekulé-style double bonds instead of the original encoding, which was already present as property on the structure after decoding and would be returned in case there are no explicit computation parameter requirements.

The fourth, optional argument is a collection of formatting bits to modify the formatting of the output value. Example:

```
select ens_string_property("c1ncccc1","E_STEREO_HASH128");
select ens_string_property("c1ncccc1","E_STEREO_HASH128",null,"enum");
```

The first example computes the stereo-specific 128-bit hashcodes and outputs it in standard format as a 32-letter hex string. The second variant uses the alternative **UUID** encoding. Note that the default output format of these functions uses only the formatting bits *compact*, *precision* and *nonull*. This is different from the **ens/reaction get** script commands, which by default also uses the *enum* flag. This is because in a database context, property data is most often required in native low-level format, not reformatted to enumerated or alternatively formatted values.

The name of the function specifies its return value datatype. This datatype is independent of the datatype of the computed property. An attempt is made to cast the original property datatype into the **SQL** function return value datatype. If that fails, the function returns **SQL NULL**. It is always possible to return any function value as string. However, a native floating point property value is rounded to an integer if the **ens_long_property()** function is used, and a string property datum returns a valid numerical value with **ens_double_property()** or **ens_long_property()** only if a string instance represents a numerical value.

The blob functions serve two distinct purposes. First, it is the suitable format for native blob data, such as structure or reaction images, or property E_MINIMOL, the compact pre-parsed structure representation which is the preferable structure format for sub- or superstructure matching. Secondly, when used with other datatypes, the output is a binary, serialized format which is recognized by other cartridge functions, especially those handling structure matching. When preparing a database for structure searching by setting up additional columns for accelerated matching, this function is used to compute binary, quick access versions of complex data not natively supported by standard **SQL** datatypes, such as short integer vectors (property E_ELEMENT_COUNT, used by **match_formula()**), or bitvectors for substructure screening (property E_QUERY_SCREEN, used by **match_substructure()**), superstructure screening (property E_NO_HYDROGEN_QUERY_SCREEN, used by **match_superstructure()**) or similarity determination (default property E_SCREEN, used by **similarity()**). These data items with hidden complexity are stored in standard database blob columns. This is explained in more detail in the chapter on database preparation.

Structure Matching Functions

There are five standard functions for matching chemical structures. These implement formula matching, substructure matching, superstructure matching and similarity computation.

Formula Matching

```
match_formula(elementcolumn/elementdata,operator,matchexpression)
```

`match_formula()` performs a formula match. This is the database equivalent of the **`ens formulamatch`** script command. The element column argument is expected to correspond to binary `E_ELEMENT_COUNT` data (computed with **`ens_blob_property()`**), either as a column when scanning a table, of as directly computed data of a test structure. The operator is a string constant and either >= (may contain elements not mentioned in the match expression), = (no elements not mentioned in the match expression) or ‘<‘ (must contain elements not mentioned in the search expression). The syntax of formula expressions is documented on the chapter on the **`molfile scan`** command. Formula expressions can contain pseudo atoms, element count expressions and other items which are not simple element stats, so their functionality goes considerably beyond simple element counting.

Example:

```
select match_formula(ens_blob_property('c1ncccc1','E_ELEMENT_COUNT'),'>=','C5');
```

This is a simple direct match check. More typical would be a query on a column which contains precomputed element vector data, like

```
select molregno from compound_query where match_formula(formula,'=','C5NH+');
```

Similarity Computation

```
similarity(screencolumn/screendata,screencolumn/screendata,?comparisonflags?,
   ?cmpval1?,?cmpval2?)
```

The **`similarity()`** function computes various similarities between bit vectors. Its default method is Tanimoto similarity. The two vector arguments can either be input as precomputed values of property `E_SCREEN` (as column reference, or computed by **`ens_blob_property()`**), or as a decodable structure identifier, such as a **SMILES** string or **SD** record. If the argument is a structure identifier, `E_SCREEN` property data is automatically computed for that structure.

The optional third argument of the function is the bitvector comparison method (see **`prop compare`** scripting command). The default is *tanimoto*, but other methods such as *tversky*, euclid, *cosine*, *dice*, *forbes*, *hamman*, *kulczynski*, *pearson*, *russelrao* and *yule* are supported. If the method supports additional weighting parameters, they can be specified as additional parameters (see again documentation of the **`prop compare`** scripting command).

The function returns an integer percentage value between 0 (no similarity) and 100 (identity of the used bitvector). The standard Tanimoto similarity value is compatible with the **PubChem** similarity value without identity boosting.

The command implicitly expects to use the `E_SCREEN` bitvector, and that is what is computed when a function argument is a structure specification. Mixing different bitvector properties as value arguments generally does not yield useful results. However, it is possible to use other bitvector properties when the values are explicitly computed (or have been pre-computed as a different property and stored in a table column), and there is a match between the bitvector property for both comparison arguments.

A typical query between a structure and a pre-computed bitvector database column for records exceeding a similarity threshold looks like

```
select molregno, similarity(simscreen,'c1ncccc1') from compound_query where
  similarity(simscreen,'c1ncccc1')>=95;
```

but is can also be used in different fashions, like for direct structure comparison

```
select similarity('c1ncccc1','c1ncccc1C');
```

which is equivalent to

```
select similarity(ens_blob_property('c1ncccc1','E_SCREEN'),
  ens_blob_property('c1ncccc1C','E_SCREEN'));
```

Above query would also work (though likely with different numeric results) if another bitvector property was specified in the property computation function. The streamlined example uses `E_SCREEN` implicitly.

Substructure Matching

```
match_substructure(screencolumn/screendata,structurecolumn/structuredata,
   substructurespecification,?ssflags?)
match_substructure_ex(screencolumn/screendata,screenproperty,screenparams,
   structurecolum,structuretype,substructurespecification,?ssflags?)
```

The **`match_substructure()`** function is used to identify structures which match a substructure.

Its first parameter is a column or constant value used for screening. In substructure matching, the actual comparison between the query structure and a database structure is comparatively expensive, so an attempt is made to filter out structures which cannot match at an early stage. A standard method for this is to use a bitvector screen. In the screen vectors, every bit corresponds to one or more structural fragments found in the database structure or query structure. For the database structures, these values are pre-computed and stored in a bitvector column. For the query structure, the same fragment set is computed once at the beginning when the query is executed.

Every fragment found in the query structure must also be present in the database structure in order to make a match possible, though this is usually (except when a query structure directly corresponds to a screen vector fragment) not a sufficient condition. The screening process is a fast operation since it only involves the test whether any set bit in the query substructure vector is also set in a database record. The standard screen property is `E_QUERY_SCREEN`. This is what the contents of the database column is expected to contain, and what is computed for the query structure. Technologically, it is a bit-folded, open fragment set, with multi-level fragment generalization, and a standard length of 1952 bits. The odd length parameter ensures that it will, in its binary, 4-bytes-segmented serialized format (see **`ens_blob_property()`**) just fit into a 252-bytes blob column, which is below the limit of 255 bytes which in some database systems is a boundary for the most compact and efficient blob storage class.

It is possible to set the screening column to **`NULL`**. In that case, the screening step is omitted. It is also possible to use a **`NULL`** structure column - in that case, only screening is performed, but no atom-by-atom match.

The second argument is the structure to match. This is also usually a column reference. The standard data format is a pre-parsed and quickly expandable binary structure specification defined by property `E_MINIMOL` which can be computed by using **`ens_blob_property()`** from some other structure encoding, such as **SMILES** or **SDF** records, which is already in the database.

The third argument is the substructure specification. This is expected to be a constant parameter. All query formats which can be decoded by the **`ens create`** script command are allowed, and it is also possible to use non-query formats such as **SMILES** with a full set of hydrogens for special purposes. Common types are **SMARTS**, **QUERYSLN** and **MDL** query **MOLFILES**, but the most powerful method to pass a query is to use a pack string of a toolkit ensemble object (see **`ens pack`** command) which can be set up by a client in any fashion. Even if it is just a simple re-packaging this can be helpful because of the difficulties in transmitting for example a binary **CHEMDRAW CDX** query file to the database uncorrupted.

The fourth optional argument is a collection of match flags. If it is not set, the default match flags combo (*bondorder*, *useatomtree*, *usebondtree*) is set. Other popular flags are *isotope*, *stereo*, *relativestereo*, *charge*, etc. - there are about 60 of these. Please refer to the documentation on the **`match ss`** script command. If the argument is used, the set flags supersede the default flag set, so please make sure to include those flags from the default set you want to keep.

A typical **SMARTS** substructure query looks like this:

```
select molregno from compound_query where match_substructure(screen,molecule,
  '[!R,H]-[#6]-1=[#6](-[!R,H])-[#6](=[O,S,N])-[#6](-[!R,H])=[#6](-[!R,H])-[#8]-1
  ')=1;
```

The match function variant **`match_substructure_ex()`** accepts additional parameters for more control over the screening and match process. First, the standard function expects the screen vector to be property `E_QUERY_SCREEN` with default computation parameters. This is what is used to compute the screen vector for the query structure. The extended function allows the naming of another screen property, and also a custom parameter set for this property. The screen value column must than contain data computed for this property and parameter set, and the same property/parameter combo is used to compute the screen for the query structure.

The database structure part is also more flexible. In addition to accepting the standard `E_MINIMOL` encoding of the database structures tested, an additional structure encoding type parameter allows the use of additional encodings. The parameter is a constant string and can be one of *minimol*, *auto*, *smiles*, *molfile*, *sln*, *cdx*, *cdxml* or *inchi*. The database structure records are then expected to correspond to the selected type. Type *auto* performs an automatic format classification on each record data item as in the **`ens create`** command and introduces another layer of potentially time-consuming analysis. Generally, decoding every structure record which passes the screening step from one of these formats other than the standard *minimol* type is time-consuming. Hover, this extended function allows the direct structure querying of databases with structure records which have not been augmented for cartridge use, and using this functionality for small tables with just a few thousand records is certainly feasible.

A simple extended substructure match on a constant structure without screening:

```
select match_substructure_ex(NULL,NULL,NULL,'c1ncccc1','smiles','c1ccccn1');
```

Superstructure Matching

```
match_superstructure(heavyatomcolumn/heavyatomdata,screencolumn/screendata,
   structurecolum/structuredata,superstructurespecification,?superflags?)
```

The syntax of the **`match_superstructure()`** function is very similar to **`match_substructure()`**. The main differences are:

There is an additional filter column as first argument, which contains the number of heavy (non-hydrogen) atoms in the database structure. Since matching database structures cannot have more heavy atoms than the query, this is used as a simple and fast first step filter. If such a column is not available, a **NULL** value can be passed.

Hydrogens in the database structures are not matched, and this also applies to the fragment filtering step. Since the standard substructure screen `E_QUERY_SCREEN` does encode hydrogen-containing fragments, a separate screen bitvector based on property `E_NO_HYDROGEN_QUERY_SCREEN` which omit hydrogen-containing fragments must be used instead. It is set up in the same fashion in database preparation as the normal screen for substructure, just with the changed property name. Be careful to use the correct screen column in the query statements - accidentally using the standard substructure screen is a common error.

A simple superstructure search looks like this:

```
select molregno from compound_query where
  match_superstructure(heavyatoms,superscreen,molecule,'c1ncccc1C')=1;
```

This query with 2-*methylpyridine* as query structure returns among other records a *pyridine* molecule, if one is contained in the database.

Full-Structure Matching

There is no dedicated function for full-structure matching. In order to perform this operation, there are two standard approaches.

The first, and recommended, method is to pre-compute one or more hashcode value columns for the database structures, and then perform a simple equality comparison between the hashcode of the query structure, and the hashcode column values.

Example:

```
select molregno from compound_query where stereohash =
  ens_string_property('c1ncccc1','E_HASHSY');
```

This is a very fast operation, especially if there is an index on the hashcode column. Computing the hashcode of the query structure takes only a few milliseconds.

Alternatively, but slower, it is also possible to perform a substructure query with a query structure which has defined hydrogens. Still, this is not completely identical to a hashcode query on the structure ensemble since the substructure would also match if the query structure was one of several components of the record.

```
select molregno from compound_query where
  match_substructure(screen,molecule,'[cH]1n[cH][cH][cH][cH]1')=1;
```

This can be fixed by setting a match flag which requires the match of the full structure ensemble:

```
select molregno from compound_query where
  match_substructure(screen,molecule,'[cH]1n[cH][cH][cH][cH]1',
  'bondorder|matchfullens')=1;
```

Preparing a Database for Structure Queries

A common task is to prepare a database which already contains some kind of computer-readable structure representation, such as a **SMILES** strings or **SD** records, for efficient structure querying by the cartridge functions. While the cartridge can work on raw **SMILES** columns etc., performance gains when using optimized query fields with pre-computed data can be many orders of magnitude.

The standard approach is to add an auxiliary table which contains the columns needed for efficient query processing by the table. The *sql* subdirectory of a standard toolkit installation contains sample **SQL** script files which perform this task for the **CHEMBL** database, which can be freely downloaded as database dumps for various database systems. The addition of a dedicated structure query table has the advantage that it does not disturb the general layout and configuration of the main database tables.

The **CHEMBL** database contains **MDL MOLFILE** records of the structures in a blob column (in table *compound_structures,* column *molfile*), which is a typical case. Any other structure representation which the toolkit can read (like **SMILES** strings or **CDX** blobs) would work as well. The query table builder statement decodes the structures from the original record and adds the derived information to the auxiliary query data table.

The exact syntax for defining the query data table varies a little from database to database. Here is the **MYSQL** version of the **CHEMBL** example:

```
create table compound_query (
screen binary(252) not null
  comment 'filled with binary property E_QUERY_SCREEN, 244 bytes screen bits plus
  4 bytes header plus 4 bytes set bit count',
molecule blob not null comment 'filled with binary property E_MINIMOL',
superscreen binary(252) not null
  comment 'filled with binary property E_NO_HYDROGEN_QUERY_SCREEN, 244 bytes
  screen bits plus 4 bytes header plus 4 bytes set bit count',
simscreen binary(120) not null
  comment 'filled with binary property E_SCREEN, 112 bytes screen bits plus 4 bytes
  header plus 4 bytes set bit count',
molregno int not null primary key
  references compound_structures(molregno) on delete cascade on update cascade,
atoms int not null comment 'atom count',
heavyatoms int not null comment 'heavy atom count',
weight float not null comment 'molecular weight',
simplehash char(16) not null comment 'filled with string property E_HASHY',
stereohash char(16) not null comment 'filled with string property E_HASHSY',
isotopehash char(16) not null comment 'filled with string property E_HASHIY',
isotopestereohash char(16) not null comment 'filled with string property
  E_HASHISY',
formula varbinary(238) not null comment 'filled with binary property
  E_ELEMENT_COUNT, max length 2*(oganesson118+1)',
index (simplehash),
index (stereohash),
index (isotopehash),
index (isotopestereohash)
);
```

There are several groups of columns:

- The reference field *molregno* links the structure query table to the rest of the data in the database. We also add a foreign key reference on this field for improved database robustness. The exact language to support automatic updating of the structure query table if the main structure table contents change is dependent on the database system - the added **SQL** syntax to handle structure record deletions on the main table is only a start. In case the main structure table does not change, or updates can be run in batch, nothing needs to be tuned.
- Three sets of bitvector screens for substructure, superstructure and similarity. These are stored in a blob column with a length less than the maximum size of the smallest blob database type (usually 255 bytes). The length of these fields is fixed because they are implied by the toolkit serialization methods. The formula to compute the proper lengths is documented in the comment.
- One blob column for the pre-parsed structures for accelerated matching. This is a field of variable length. Specifying a maximum column size of 1K bytes is sufficient for standard drug-sized molecules, if the database can perform optimizations with this additional information. However, for normal set-ups we do not recommend explicit sizing of this field for maximum flexibility, and for coping with the occasional huge molecule.
- One blob column for the element count vector for formula searches. This is a column of variable length since the element count vector is truncated after the element with the highest number found in the structure.
- Four hashcode columns for quick full-structure search with or without stereochemistry, and with or without isotope labels. The toolkit provides a collection of additional hash codes for other full-structure comparison methods, which could be added if desired. For maximum performance, an index is defined on these columns. The example uses the 64-bit hashcodes, which is considered sufficient for databases with less than 50 million compounds. Beyond that, the 128-bit hashcode versions are suggested. For the sake of simplicity, the hashcodes are stored in fixed-length string fields. It is also possible to use integer encoding (or a native database **UUID** datatype for the 128-bit variants), but be careful to ensure the use or unsigned integer datatypes and proper handling of the comparison of integers of different byte length in **SQL** statements if you go this route. In our experience, string comparisons are more robust, and with the indices there is negligible performance difference.
- A couple of additional useful simple structure properties, like atom count, heavy atom count (this helps with superstructure searches) and molecular weight. More could be added if desired.
- The order of the fields is not important. Some database systems access the first field slightly faster, especially if the column set contains variable-length fields. If that is the case, we recommend to put the substructure screen vector into that position, as in the example.

The statement to fill the structure query table from the main structure table with the **CHEMBL** example is:

```
insert into
  compound_query(screen,molecule,superscreen,simscreen,molregno,atoms,heavyatoms
  ,weight,simplehash,stereohash,isotopehash,isotopestereohash,formula)select
  ens_blob_property(molfile,'E_QUERY_SCREEN'),
  ens_blob_property(molfile,'E_MINIMOL'),
  ens_blob_property(molfile,'E_NO_HYDROGEN_QUERY_SCREEN'),
  ens_blob_property(molfile,'E_SCREEN'),
  molregno,
  ens_long_property(molfile,'E_NATOMS'),
  ens_long_property(molfile,'E_HEAVY_ATOM_COUNT'),
  ens_double_property(molfile,'E_WEIGHT'),
  ens_string_property(molfile,'E_HASHY'),
  ens_string_property(molfile,'E_HASHSY'),
  ens_string_property(molfile,'E_HASHIY'),
  ens_string_property(molfile,'E_HASHISY'),
  ens_blob_property(molfile,'E_ELEMENT_COUNT')
  from compound_structures;
```

Decoding the structure separately for each field looks inefficient, but thanks to the structure cache this does not really happen and every structure in the *molfile* blobs of table *compound_structures* is only decoded once.

The **CHEMBL** database contains about 1.8 million structures. On a normal but reasonably powerful developer PC or server filling the full structure query table takes about 7 hours in single-threaded operation.

Useful Tips and Tricks

When debugging cartridge queries and query results, it is often useful to test these queries and the command line. Unfortunately, result tables which contain **SMILES** or **SD** records are not easy to interpret as raw **ASCII** strings. Some creative use of cartridge functions can help you with this.

```
select ens_blob_property('c1ncccc1','E_GIF') into dumpfile
  '/home/someuser/pyridine.gif';
```

Above statement (for the **MYSQL** database) directly produces an image from a structure which can be inspected with any image viewer. Depending on the database set-up, writing *dumpfiles* may be restricted to specific directories or users. In this case, we use a constant **SMILES**, but of course this can be a structure specification from a query. There can be only a single *dumpfile* record, so a **`limit 1`** restriction on the query is advisable if there could be multiple results. Normal image generation parameters can be set, as shown below:

```
select ens_blob_property('c1ncccc1','E_GIF',
  'bgcolor black atomcolor element height 200 width 200 format png bondcolor
  white') into dumpfile '/home/someuser/pycolor.png';
```

Script Functions

A unique feature of the CACTVS cartridges is that they support scripting of toolkit code on the database server.