DRAFT CHAPTER OF THE OFFICIAL PyMOL MANUAL (For PyMOL Sponsors)

A new installment

This chapter is part of a comprehensive manual-in-progress, so you will find references to chapters that are not included here. Nonetheless, we think you may find it helpful, especially if you are a new user. Please email help@schrodinger.com if you find it in any way confusing or incomplete.

Only for sponsors

This is an incentive product for PyMOL sponsors. Please do not post it publicly or otherwise share it with the general public. Incentive products, such as this manual, are exclusively for sponsors, and sponsors are what make possible PyMOL's continued development, documentation, and support. To confirm or inquire about sponsorship, please email sales@pymol.org.

Chapter TC. Typing Commands

Chapter HTGR2 presents a few typed commands to help new users get started. This chapter gives a more thorough view of the range of PyMOL Commands, and Chapter ES explores the details of Selection Expressions. This information corresponds to **PyMOL version 1_2r1.**

Two other resources are indispensable for working with typed commands. The first is the PyMOL help utility, accessed through the command line. Type `help` *or* `help command` *for a full list of PyMOL commands. Choose a command from the list that appears in the command history area and type* `help <command-name>` *to get information (see Figure TC.1). The other great resource for information on commands is the set of reference pages for PyMOL sponsors at http://pymol.org/id/command. For settings, in addition to the reference pages at http://pymol.org/id/setting, the PyMOL Wiki provides a growing compendium of expanded explanations. Commands and settings that are not explained in this chapter can be found in these resources.*

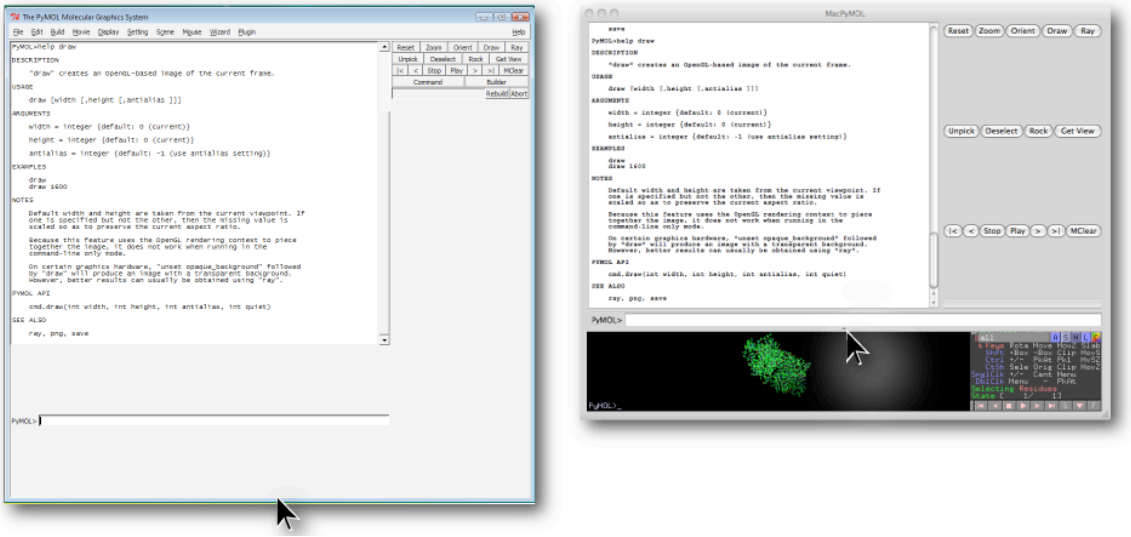


Figure TC.1 Help on PyMOL's commands is shown in the command history and diagnostics area. To enlarge that area on a Windows PC (left), place the cursor at the bottom of the Tk window and drag down. On the Mac, place the cursor on the dot below the command line (shown at the tip of the cursor on the right) and drag down.

Contents

Keywords 6

Selection-expressions and name patterns 7

Commands with Default Arguments 8

Menu-equivalent Commands 10

Representation 11

Scene-manipulation 14

Measurement 15

Structural comparison 18

File input and output 20

Settings 23

Command Line Features 27

Command completion using TAB 27

Command inference 28

Command argument inference 28

File name completion using TAB 28

Climbing the stack of commands 29

Viewing commands in the Display Area 30

Keywords

A typed PyMOL command always starts with a keyword, a word that calls PyMOL to execute an action. It ends when you hit `enter` or `return` on your keyboard. PyMOL commands can be typed into a command line in a PyMOL window (see Figure TC.2), and into scripts that PyMOL can read and execute.

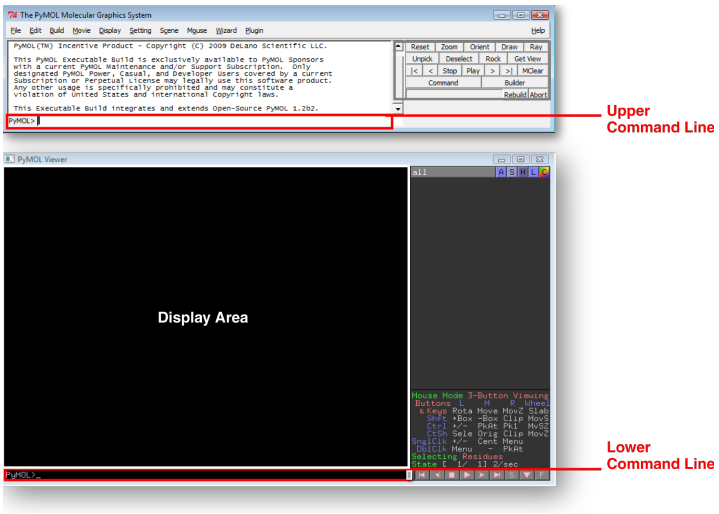


Figure TC.2 PyMOL's upper command line is useful when you are tracking the command history in the window just above it. The lower command line is more convenient when your eyes are engaged in the Display Area.

The simplest commands consist of a keyword alone. For example, typing

```
PyMOL> quit
```

will end your PyMOL session and close the program. The `quit` command never takes an argument. (An argument is just one or more keywords that PyMOL uses to complete a command.)

Other single-keyword PyMOL commands correspond to menu choices or buttons on a PyMOL window (see Figure TC.3). For example, typing

```
PyMOL> reinitialize
```

causes PyMOL to delete all objects and restore the default program settings, as does clicking the **reinitialize** button in the top bank of buttons.

In Chapter HTGR1, we reviewed several ways to deselect the current selection. Typing

```
PyMOL> deselect
```

is another way to disable all enabled selections. Typing

```
PyMOL> reset
```

causes PyMOL to restore the view of a molecule that it originally displayed when the coordinates were first loaded. Typing

```
PyMOL> full_screen
```

causes the PyMOL window to fill you computer screen. This command works as a toggle switch. To return to the previous display size, type `full_screen` again.

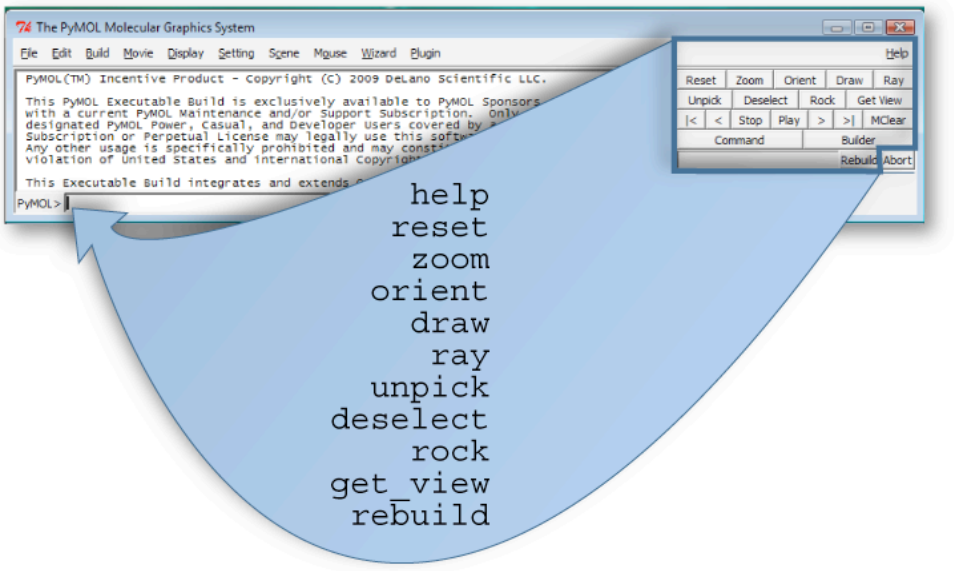


Figure TC.3 The same commands given by pointing and clicking on buttons can be typed in the command line. Many of the button commands have no arguments. Others have default arguments.

Selection-expressions and name patterns

A selection-expression is one of the two types of command argument that specify the object of a command. Several selection-expressions were introduced in the Typing Commands section of Chapter HTGR2. For example, `1uwh`, `resi 100, polymer and resi 100, resn glu+asp,` and `chain b+d+e` are all selection-expressions. Selection-expressions always refer to atoms, though they may use molecular object names. When a selection-expression contains a list with more than one item, no spaces are allowed, as in `resn glu+asp,` and `chain b+d+e.`

A different type of command argument, called a name pattern, is used to specify data files, objects (such as measurements), and parts of objects (such as bonds) that are not atoms. For example, the file name `1uwh.pdb` and the measurement object names `measure01` and `measure03` do not specify atoms, so they belong in the category of

name patterns. When there is more than one item in a name pattern, the items are separated by spaces, as in

```
PyMOL> fetch 1t45 1t46
```

Usually, a selection name or selection keyword is present in a selection-expression, signaling to PyMOL that it must select atoms. Otherwise, you can ensure that PyMOL treats a command argument as a selection-expression by enclosing it in parentheses, as in

```
PyMOL> color white, (symbol C)
```

The parentheses are optional, but they help to distinguish selection expressions from name patterns and other types of command arguments.

Commands with Default Arguments

Many commands have default arguments, so you can type the command keyword alone and PyMOL will supply the rest. For example, the default argument to `zoom` is the selection-expression `all`. When you type

```
PyMOL> zoom
```

and hit `return` or `enter`, PyMOL scales and translates (without rotation) the enabled representations to display `all` in the Display Area. If you want PyMOL to zoom in on a specific set of atoms, as in Figure TC.4, add a selection-expression argument to the command. For example,

```
PyMOL> zoom chain b and organic
```

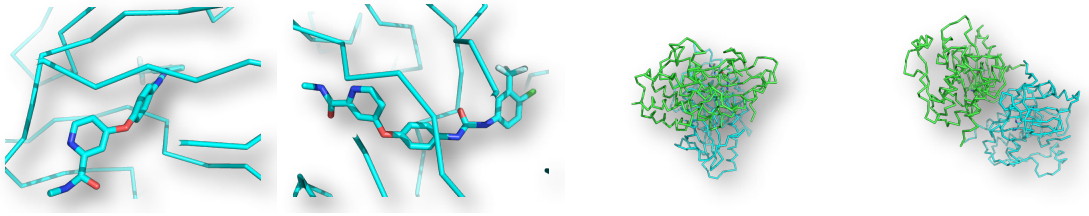


Figure TC.4 The `zoom` command (with no argument) maintains the rotation of objects, while including as many of its atoms as possible in the view. The `reset` command rotates the view back to its original rotation.

Table TC.1 gives the most familiar commands that have default arguments, along with descriptions of what they do. In most, but not all, cases, the default arguments are selection-expressions.

Table TC.1 Commands with default arguments

Keyword	Default Argument	Result
`center`	`all`	translates the Display Area, the clipping slab, and the origin to a point centered within the specified atoms
`bg_color`	`black`	changes the color of the Display Area background
`color`	`all`	changes the color of the specified atoms
`disable`	`all`	turns off the display of the specified atoms
`draw`	current size of Display Area	creates an image in the Display Area that is not ray-traced
`enable`	`all`	turns on the display of the specified atoms
`get_view`	destination of output	returns and optionally prints out the current view information in a format that can be embedded into a command script, and can be used in subsequent commands to `set_view`
`h_add`	`all`	adds hydrogens onto a molecule
`help`	all PyMOL commands	prints out help in the command history window
`hide`	`all`	turns off atom and bond representations
`orient`	`all`	aligns the principal components of the specified atoms with the XYZ axes
`origin`	`all`	sets the center of rotation
`ray`	current size of Display Area	creates a ray-traced image in the Display Area (this can take some minutes, depending on the complexity of the image)
`show`	`all`	turns on representations for the specified atoms

`zoom`	`all`	scales and translates the enabled representations in the Display Area to display the specified atoms

The command `get_view` is a useful alternative to saving scenes when you are interested in returning only to the position of the camera, and not to all the other characteristics of the display. For more information, see the online documentation.

The default argument `all` is a selection-expression. In the Selecting Atoms section, we will explore other selection-expressions.

Some keywords require one argument and supply another argument by default. For example, the keyword `color` requires one argument, the `color-name`. If you do not specify which atoms to color by adding a selection-expression, PyMOL supplies the default selection-expression `all`.

SYNTAX:	`color color-name` `color color-name, selection-expression`
EXAMPLES:	`PyMOL> color red` `All` representations are colored red.
	`PyMOL> color red, name ca` Only the representations of atoms named C-alpha are colored red.
	`PyMOL> color red, 1uwh` In this case the selection-expression is the name of an object, and every atom in the input file 1uwh.pdb is colored red.

When you type a command that has more than one argument, a comma must separate the arguments, as in

```
PyMOL> color red, 1uwh
```

A complete listing of color names defined in PyMOL is found at http://pymolwiki.org/index.php/Color_Values.

Menu-equivalent Commands

Representation

In the **Making and Representing Selections** section of Chapter HTGR1, we showed how to use menus to control the representations of objects & selections. Most of the menu choices have typed command equivalents (see Figure TC.5). These typed commands have two arguments. The first is the form of the representation, and the second is a selection-expression.

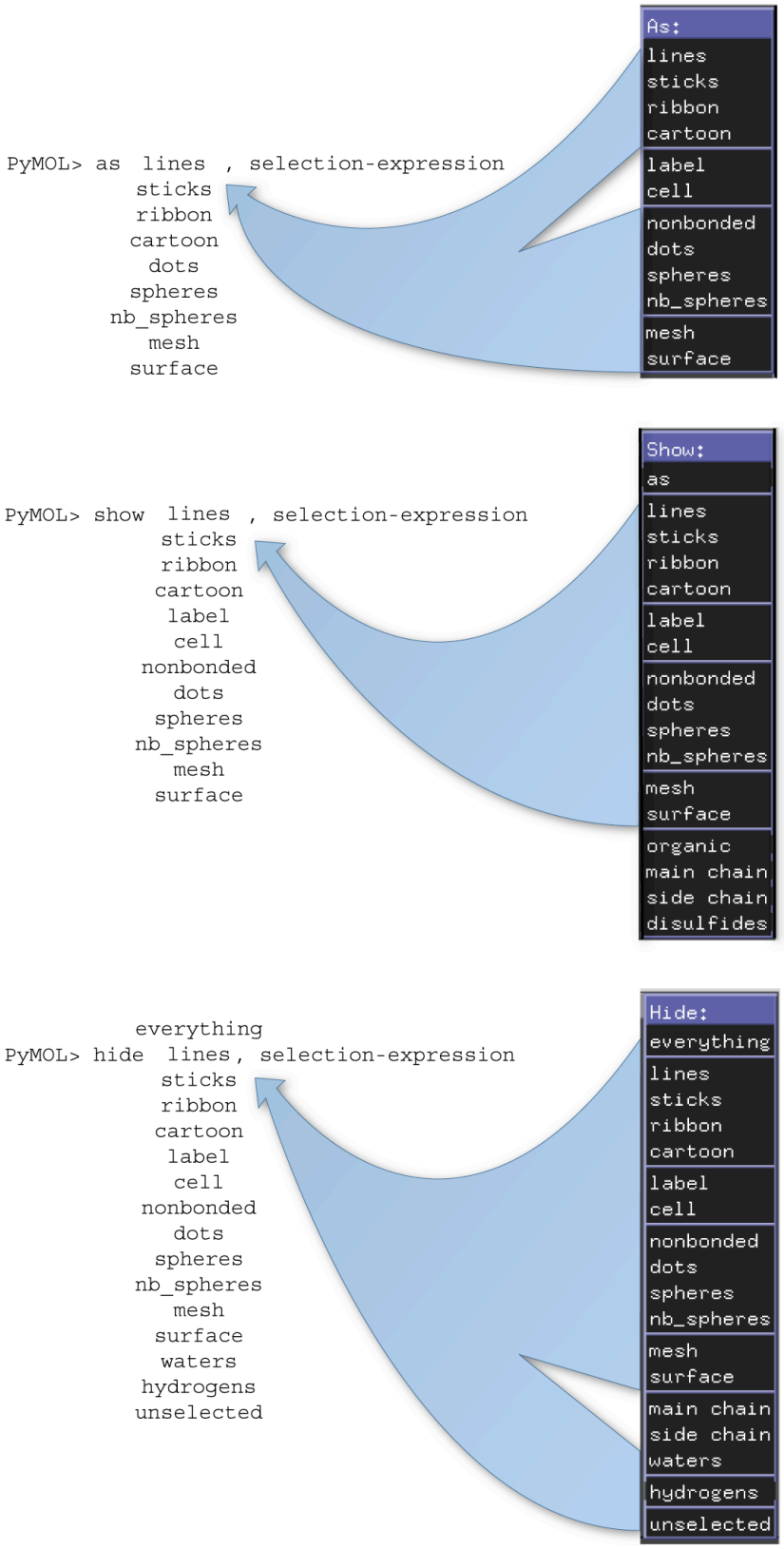


Figure TC.5 Several typed commands are equivalent to the menu choices that pop-up next to the names of objects & selections. The selection-expression in the command is just the name of the object or selection.

Figure TC.6 demonstrates typed representation commands, starting with named objects & selections from Figure X.16. A selection-expression can be the name of a named object or selection, or it can be something more complicated (see Chapter ES).

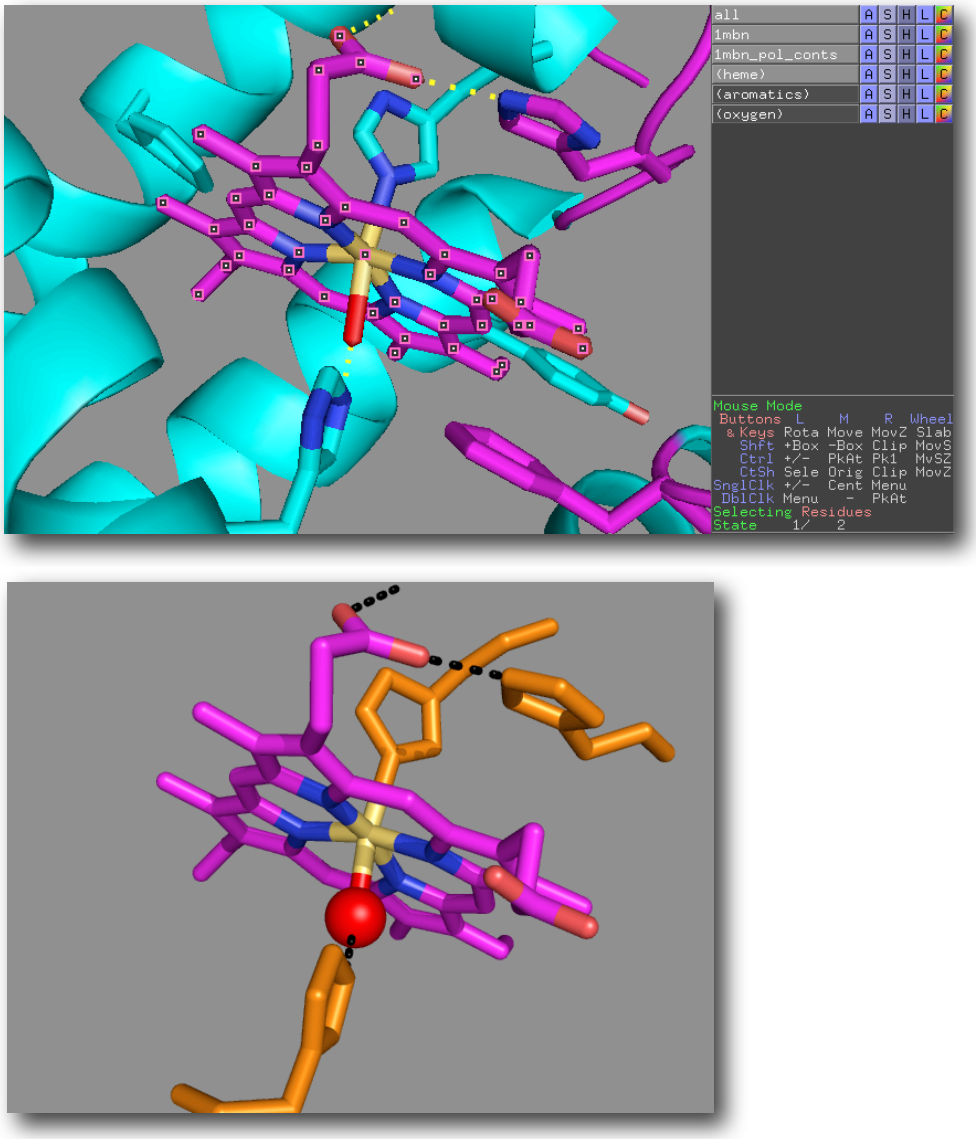


Figure TC.6 Starting from Figure X.16 (top), the following typed commands create the bottom image:

```
PyMOL> hide cartoon
PyMOL> color orange, aromatics
PyMOL> color black, 1mbn_pol_conts
PyMOL> show spheres, oxygen
PyMOL> set sphere_scale=0.5
PyMOL> hide sticks, aromatics and not resn his
```

Manipulate the image with the mouse to show the bonds to the oxygen and heme.

```
PyMOL> ray
```

Scene-manipulation

Scenes were introduced in the Save As You Go section of Chapter HTGR1, using Menu choices to give commands. Scenes can also be created and manipulated by typing. To store the current representation in the Display Area as a Scene, type

```
PyMOL> scene new, store
```

PyMOL automatically assigns a name to to the Scene as it stores the representation. The default naming system is simply a numeric series, starting with `001`. As you continue to add Scenes, they will be assigned numeric names in the order of creation.

Use the keyword `insert_after` or `insert_before,` and a Scene name (`001`, `002`, etc.) , to place the representation in the Display Area anywhere in a series of two or more Scenes, as in

```
PyMOL> scene new, insert_before, 002
```

Any Scene can be updated to match the view currently being displayed by typing the Scene name as the first argument and `update` as the second, as in

```
PyMOL> scene 001, update
```

If you are updating the last Scene created, you can use the keyword `auto` in place of the Scene name, as in

```
PyMOL> scene auto, update
```

You can also add a note to the current Scene by typing in the command line. For example, typing

```
PyMOL> scene auto, update, Annotation information
```

will display "Annotation information" the next time the current Scene is recalled.

To display a stored Scene (Scene `001`, for example) type

```
PyMOL> scene 001, recall
```

using the assigned Scene name, `001`, as the first argument, and the action, `recall`, in this case, as the second argument.

See the online documentation for a complete description of Scene commands.

Measurement

Distance measurements To determine the distances between atoms, you can use PyMOL's Distance Wizard, described in Chapter HTGR2, or you can type the `distance` command with two selection-expression arguments. For example, typing

```
PyMOL> distance 17/nh2, 6/cz
```

will create a distance object. (By default, the object will be named **dist01** for the first distance object you create in a session and **dist02** for the second, etc.) The distance object name and menu will be displayed in the objects & selections area, and a line between specified atoms, labeled with the distance in Ångstroms, will be added to the Display Area, as shown in Figure TC.7.

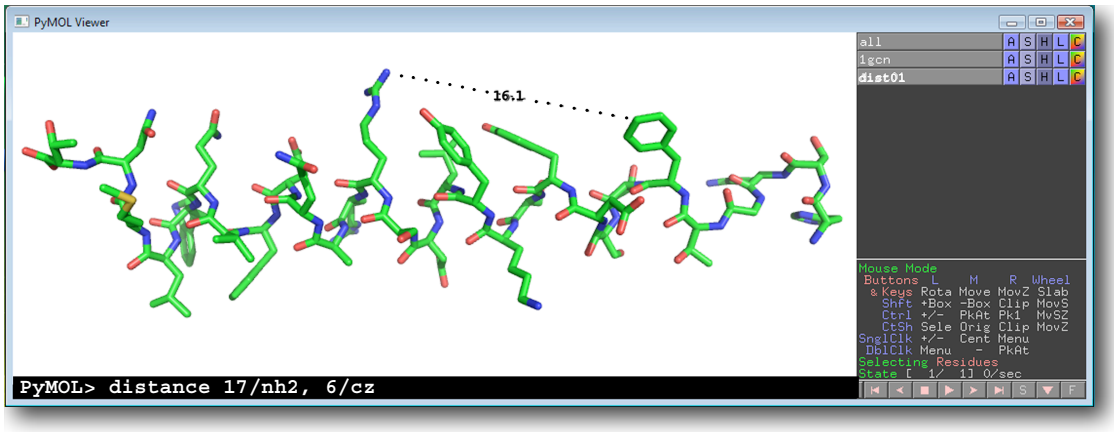


Figure TC.7 An anonymous distance command is given in the lower command line, resulting in a distance object named dist01, which is displayed in the objects & selections list.

If you want to name distance objects differently, type the name as the first argument to the `distance` command, as shown in Figure TC.8.

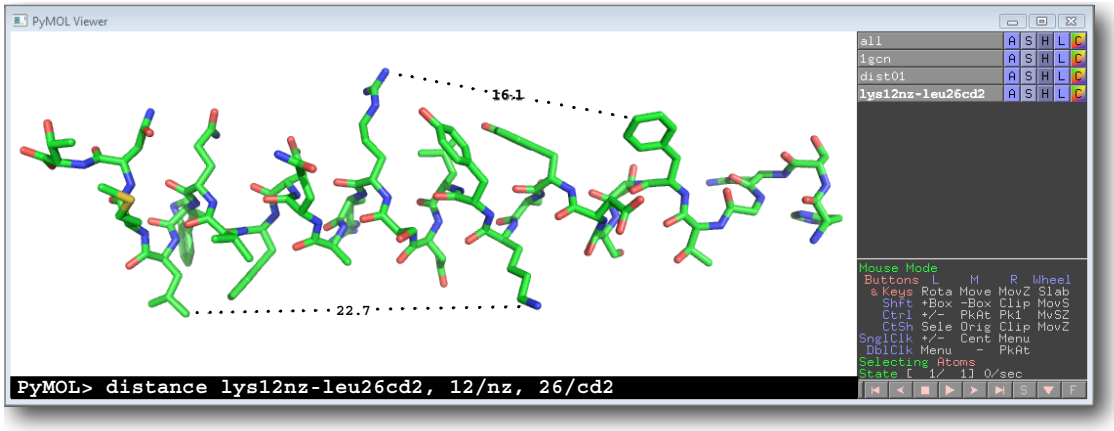


Figure TC.8 A named distance command is given, and the object takes the given name.

The distance command works for atoms, residues, chains, and even objects, creating a measurement object that includes a distance line and label for every possible pair with one atom from each of the two selection-expressions. This may be more information than you want, so you can provide a cutoff distance, restricting the measurements to atoms no further apart than the cutoff. Figure TC.9 shows that

```
PyMOL> distance prot-lig-dists, ligand, protein, 3
```

will create a distance object called prot_lig_dists that contains only atom pairs within 3 Å of each other.

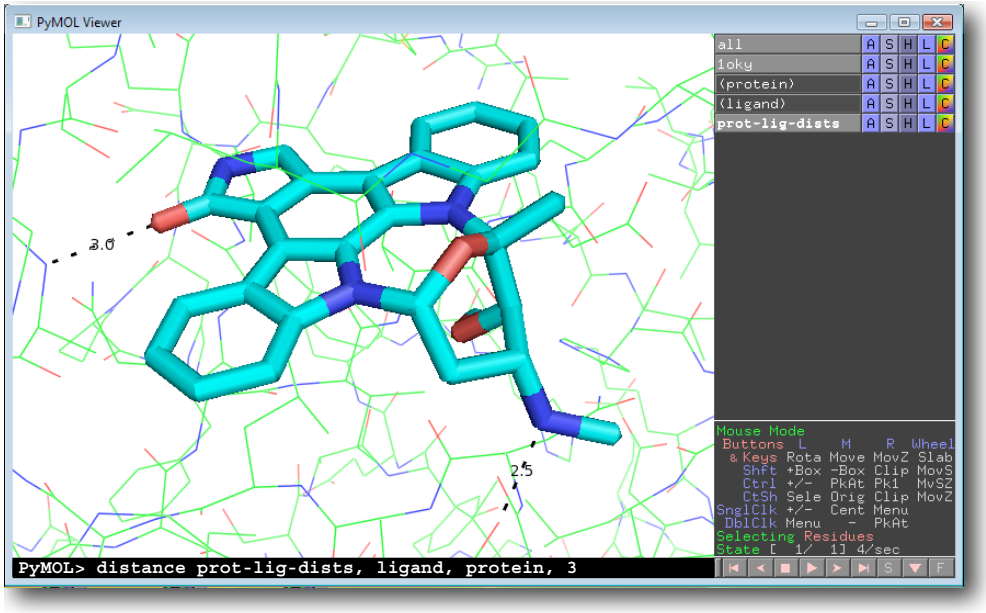


Figure TC.9 A named distance command is given with a final argument that limits the distances separating the pairs that are returned in the distance object.

You can set a mode argument to determine the types of atom pairs that are measured. For example, typing

```
PyMOL> distance prot_lig_all, protein, ligand, mode=1
```

will measure *bond* distances between the objects or selections named **protein** and **ligand**. Type `mode=0` to create a distance object containing all interatomic distances (between bonded and nonbonded pairs), or `mode=2` to measure only polar contact distances. Combining a cutoff argument with `mode=2` is useful for examining putative hydrogen bonds, as in

```
PyMOL> distance hbonds, all, all, 3.2, mode=2
```

Angle and dihedral measurements Unlike the `distance` command, the `angle` command requires an angle object name as its first argument. The following three arguments select the atoms that define the `angle`, as in

```
PyMOL> angle required_name, 93/ca, 93/cb, 93/cg
```

Figure TC.10 shows the angle object that appears in the object & selections list and the representation of the angle (in blue) that appears in the Display Area.

Similarly, the `dihedral` command requires, as its first argument, a name for the dihedral object PyMOL will create. The following four arguments select the atoms that define the `dihedral`, as in

```
PyMOL> dihedral heme-his-dih, 93/ce1, 93/ne2, 155/fe, 55/na
```

Figure TC.10 show the corresponding `dihedral` object.

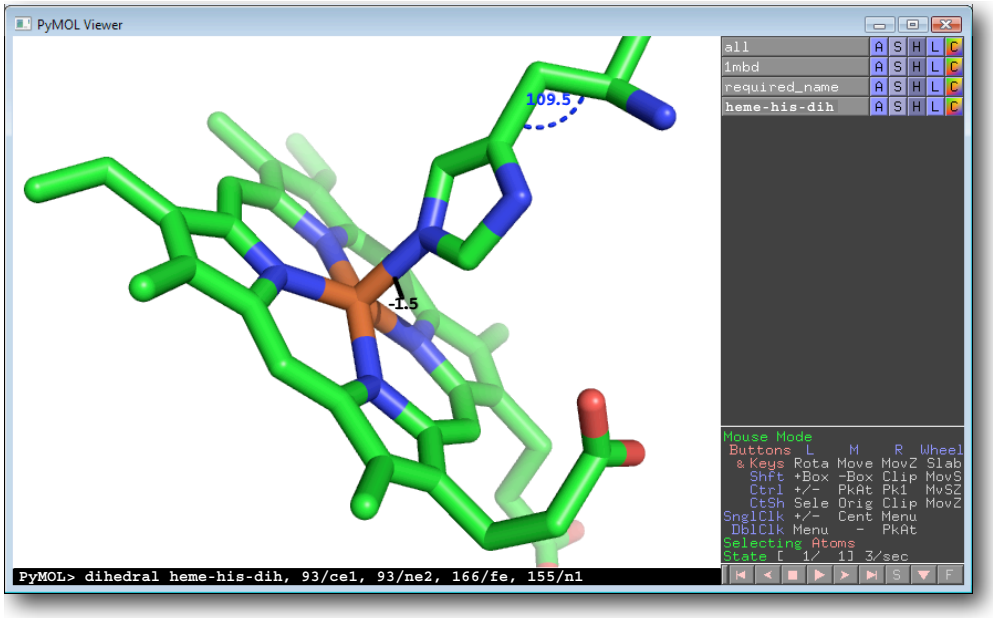


Figure TC.10 A named dihedral angle command is given with the three arguments that define the dihedral.

Structural comparison

Aligning structures Chapter HTGR2 showed how to use the **A** Action menus that invoke PyMOL's sequence and structural alignment routines to compare structures, and how to use the Pair Fitting Wizard to produce alignments based on specific atom pairs. The typed `pair_fit` and `align` commands were introduced there, as well. Typed alignment commands give you more control than the menu alignment commands

because they allow you to choose the atoms for the computations. Typed pair fitting commands give you maximum control because they allow you to choose any number of specific atom pairs.

Figure TC.11 compares the alignment of **1z5m** with **1t46** using (left) C-alphas from the entire chains and (right) C-alphas from two specific beta-strands. Alignment across the entire chains is typed as

```
PyMOL> align 1z5m////ca, 1t46////ca, object=aln_name
```

This corresponds to the menu-based alignment command. An alignment based on C-alphas of the specific range of residues is typed as

```
PyMOL> align 1z5m///85-100/ca, 1t46///592-607/ca, object=a_name
```

The third argument, `object=object_name`, is optional. It causes PyMOL to display yellow cgo_lines connecting the atom pairs used in the computation, to grey out the atoms that are not aligned in the Sequence Viewer, and to add the named alignment object and its menus to the list of objects & selections (so you can show or hide the cgo_lines).

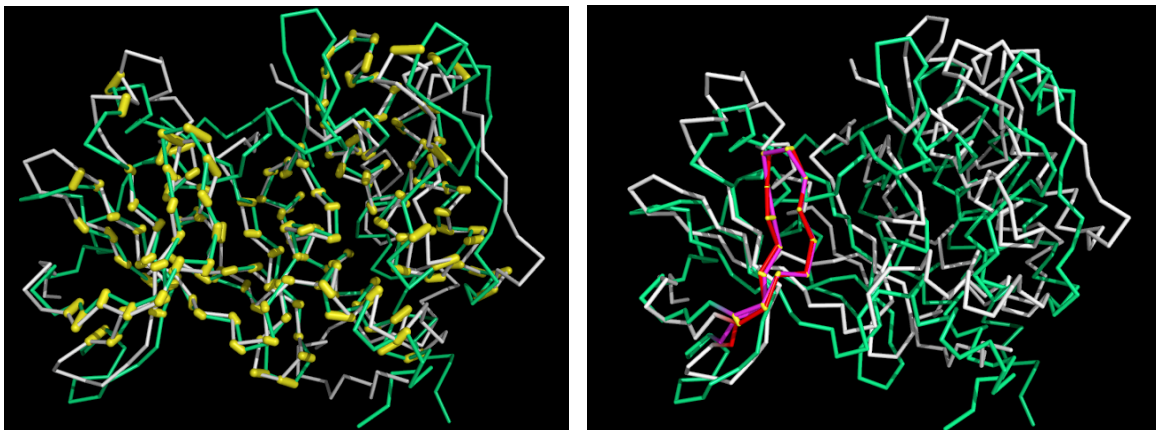


Figure TC.11 The typed alignment command allows you to specify a set of atoms for computing the alignment. (left) `PyMOL> align 1z5m, 1t46, object= aln_name`;
(right) `PyMOL> align 1z5m///85-100/ca, 1t46///592-607/ca, object=a_name`
In these figures, the cgo_line_radius was set to 0.35, and the ribbon_radius was set to 0.2 (see the Settings section).

The two objects in the alignment need not contain the same number of atoms. PyMOL's alignment routine begins with a dynamic-programming protein sequence alignment, which passes only matching atoms on to the structural alignment computation.

Pair-fitting structures The `pair_fit` command requires you to select at least two different pairs of atoms (or pairs of selection-expressions) as a basis for superposing two molecular objects.

As noted in Chapter HTGR2, the `pair_fit` command requires that each of the paired selections contains the same number of atoms. Pair fitting does not produce a `pair_fit` object, but simply computes a pairwise RMS fit, superposes the structures in the Display Area, and reports the RMSD of the fit in the command history area.

For example, Figure TC.11 shows the `pair_fit` of two transfer RNAs, 1yfg and 3cw5, resulting from the typed command

```
PyMOL> pair_fit (1yfg and resid
2+7+12+18+21+31+36+41+51+56+61+66+71 and name P), (3cw5 and resid
2+7+12+18+21+31+36+41+51+56+61+66+71 and name P)
```

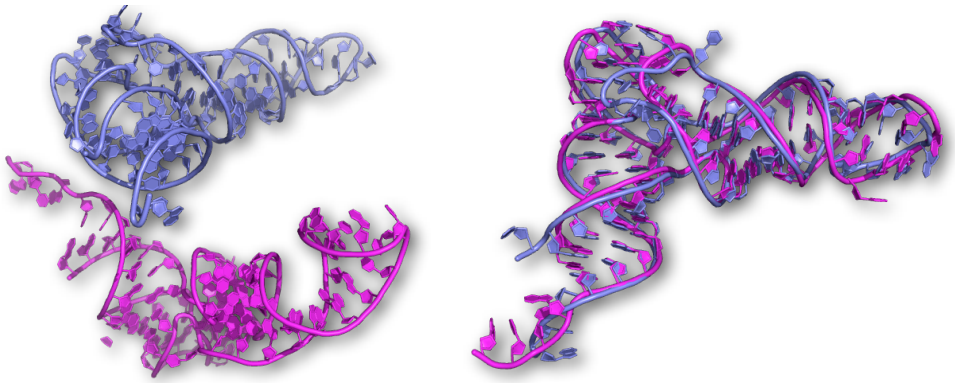


Figure TC.11 Cartoon structures of tRNAs 1yfg and 3cw5 (left) were superposed (right) using the `pair_fit` command above. The cartoon style was selected by typing
`PyMOL> set cartoon_ring_mode, 3`
(See the Settings section).

File input and output

To save or import a file in any program, you need to choose the location where the files are stored. In PyMOL, you can navigate to files using the **File** menu, or, if you are familiar with Unix, you can enter the Unix navigation commands `cd`, `pwd`, `ls`, etc., in the PyMOL command lines, and use typed commands for input and output.

Input The `fetch` command we have been using stores files in the user's home directory. When PyMOL is launched, the directory is set to be the user's home directory, so you can simply type, for example,

```
PyMOL> load 1yfg.pdb, yeast-t
```

to open a data file that resides there. The argument **yeast-t** is an optional object name, which PyMOL then assigns to the object. The comma between the arguments keeps PyMOL from interpreting them as a name pattern, and looking for a file named `yeast-t`. Without the second argument, as in

```
PyMOL> load ~/1yfg.pdb
```

PyMOL gives the loaded object the name of the data file, **1yfg** in this case.

A PyMOL installation creates directories, usually aliased with the pathname $PYMOL_PATH, in which data, examples, and other resources are stored. Users may find it convenient to use and store data in those directories. For example, the command

```
PyMOL> load $PYMOL_PATH/test/dat/pept.pdb
```

will locate the pept.pdb file and load it with the object name **pept**, and

```
PyMOL> load $PYMOL_PATH/test/dat/pept.pdb, test
```

will load the same file and name the object **test**.

If you choose to `cd` to another directory, subsequent input and output commands will originate from the current directory. The following input and output commands work with files in the current directory.

In addition to fetching and loading data files, you can type the command to run a script, as in

```
PyMOL> @ ./my_script.pml
```

Output Scripts can be tested and recorded by creating logs of commands during a PyMOL session. To create and open a log file called `my_log`, for example, type

```
PyMOL> log_open ./my_log
```

Typing

```
PyMOL> log_close
```

will close any log file that is open.

The typed command, `save`, is used to save sessions, molecules, and single images. The first argument to any `save` command is the path and name of the file to be saved. The file name must be complete with its file name extension because PyMOL uses it to determine which kind of file to create. For example, to save a session file, type

```
PyMOL> save ./my_session.pse
```

To save a show file, type

```
PyMOL> save ./my_show.psw
```

(Recall that session and show files can be interconverted by trading file name extensions.)

Saving the coordinates of a molecule requires an additional argument, the selection (or object) to be saved. For example, to save the coordinates for **pept** in PDB format, type

```
PyMOL> save ./pept.pdb, pept
```

Images may be saved in PNG files by typing, for example,

```
PyMOL> save ./my_image.png
```

A more powerful typed command for saving images is `png`. The basic `png` command is

```
PyMOL> png ./my_image
```

The brief `save` and `png` commands create images the current size of the Display Area. By adding arguments to the `png` command, you can control the size of the image. For example, typing

```
PyMOL> png ./my_image, 1200, 800
```

will produce an image 1200 pixels wide and 800 pixels high. If you only specify the width, PyMOL will produce an image with the given width, and scale the height to match the proportions of the current Display Area. Or, to produce an image the same size as the current Display Area, but with a specified resolution, type the argument `dpi=` with the desired number of dots per inch, as in

```
PyMOL> png ./my_image, dpi=300
```

An additional optional argument is `ray=. ray=1,` as in

```
PyMOL> png ./my_image, 1200, 800, ray=1
```

prompts PyMOL to ray trace the image before writing the file to disk, and `ray=0,` as in

```
PyMOL> png ./my_image, 1200, 800, ray=0
```

prompts PyMOL to write an image file that has not been ray traced,

Settings

PyMOL is almost infinitely customizable, with hundreds of settings accessible to the user. The **Display** menu, as well as the **Setting** menu, offer frequently used settings for point-and-click manipulation. Choosing **Setting** / **Edit** also calls up a graphical interface for editing settings. In addition, you can manipulate settings with typed commands. Typed equivalents for settings available from the **Display** menu are given in Figure TC12. Several settings from the **Setting** menu are illustrated in the figures that follow.

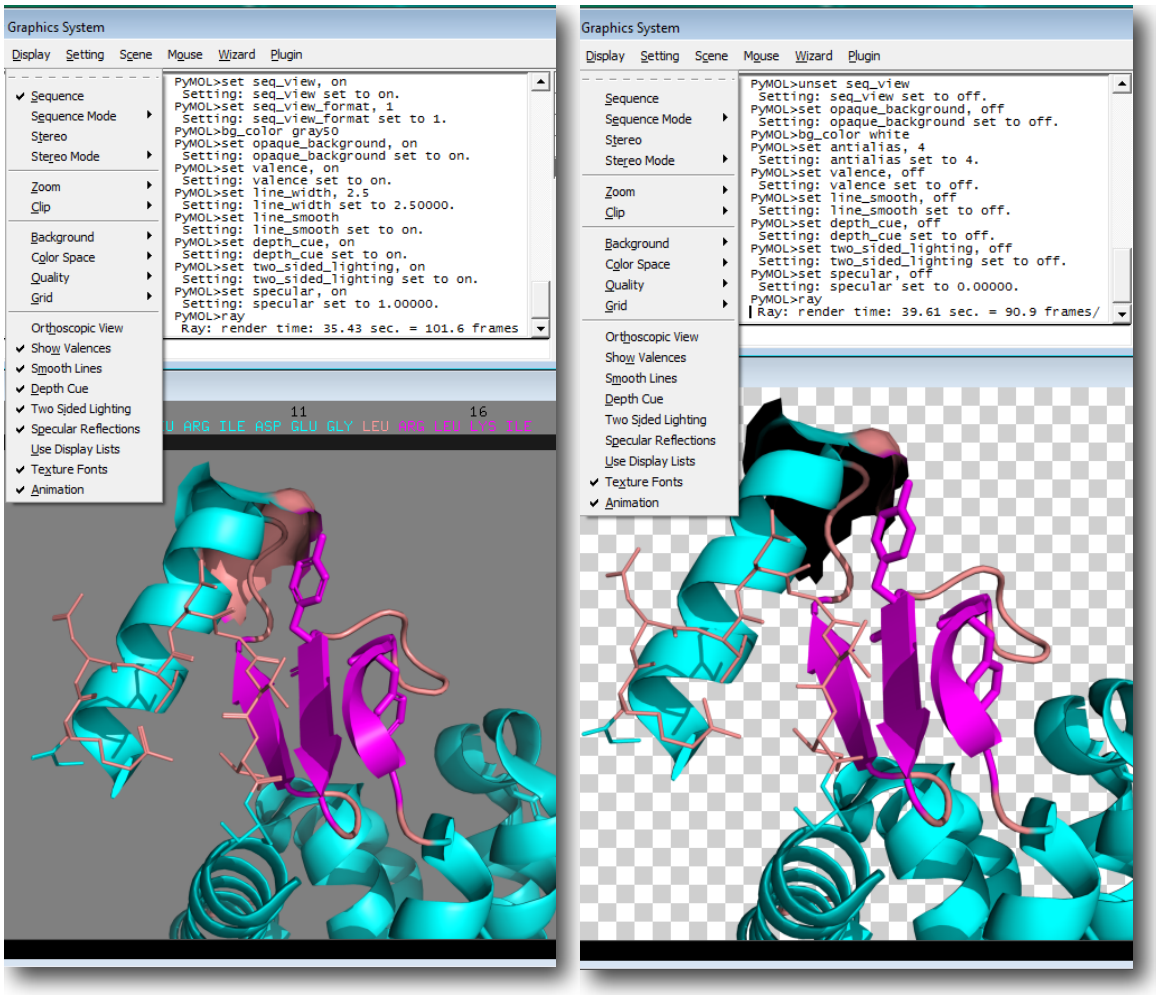


Figure TC. 12 (Left) Several settings from the **Display** menu are set to on by typed commands, which are echoed in the command history and diagnostics area. (Right) These same settings are set to `off`, or changed to contrasting values. The checkered background in the ray-traced display is an indication that the background in the PNG file is transparent.
The related settings, `seq_view_format`, `opaque_background`, `line_width`, and `antialias`, and the `bg_color` command are also shown. Descriptions of the individual settings are given in the PyMOL Wiki (seq_view, seq_view_format, opaque_background, antialias, valence, line_width, line_smooth, depth_cue, two_sided_lighting, specular), and in Appendix-Settings-ToCome.

Settings are commanded by typing three keywords: `set`, `unset`, and `get`, where `set` changes the value of a setting, `unset` returns the setting to its default value (the value PyMOL opens with), and `get` displays the current value of the setting. Some `set` commands are toggle switches. That is, you can type `set` to switch from the default value to its alternative, and `unset` to switch back to the default. For example,

```
PyMOL> set auto_show_spheres
```

changes the default `off` value of `auto_show_spheres` to `on`, causing PyMOL to display all objects as spheres when they are loaded. Typing

```
PyMOL> unset auto_show_spheres
```

reverts `auto_show_spheres` to its default value of `off`, and all subsequently loaded objects are displayed as lines. To determine the value of `auto_show_spheres`, type

```
PyMOL> get auto_show_spheres
```

and the value will be displayed in the command history window. Figure TC.12 shows other settings that are `set` from default values of `off` to `on`.

Figure TC.12 `Cartoon_cylindrical_helices` and `cartoon_fancy_helices` are `off` by default, while `cartoon_round_helices` is `on`. In the **Setting** menu and submenus, **Cartoon**, **Ribbon**, **Surface**, **Transparency**, and **Rendering**, checks appear next to settings when they are `on`.

Figure TC.13 shows a few settings that are `set` from default values of `on` to `off`.

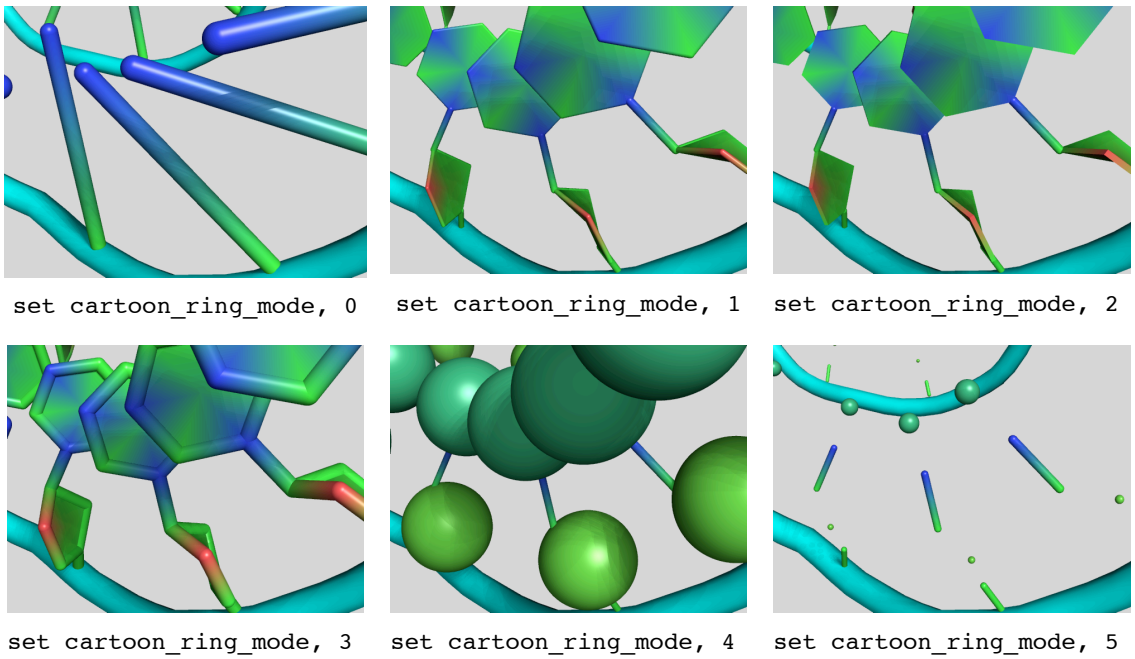
Other `set` commands require two arguments: the name of the setting and the value you wish to set. For example, cartoon representations of nucleic acid bases are available in several modes, chosen by typing

```
PyMOL> set cartoon_ring_mode, 1
```

`set cartoon_flat_sheets, off` | `set cartoon_fancy_sheets, off` | `set ray_trace_fog, off`

Figure TC.13 Note that, when `cartoon_fancy_sheets` is `off` (middle), no arrowheads are drawn to indicate the N to C direction of the chain, and that when `ray_trace_fog` is `off` (right), the background parts of the chain are just as intensely colored as the helix in the foreground of the ray-traced image. Fog, or fading of the more distant parts of the image, is one of PyMOL's graphical subtleties for giving three-dimensional depth to images.

`Cartoon_ring_mode` can be set to values of 0 through 5 (see Figure TC.14).

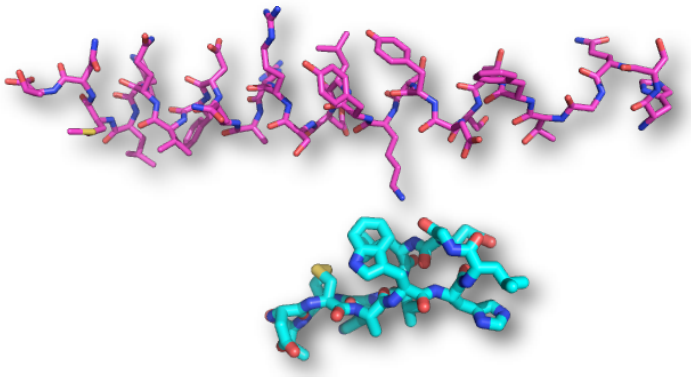


`set cartoon_ring_mode, 0` | `set cartoon_ring_mode, 1` | `set cartoon_ring_mode, 2`

`set cartoon_ring_mode, 3` | `set cartoon_ring_mode, 4` | `set cartoon_ring_mode, 5`

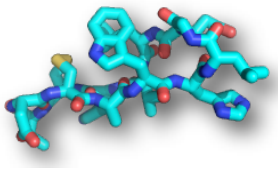
Figure TC.14 `cartoon_ring_mode 2` differs from `cartoon_ring_mode 1` in the treatment of the edges of the rings. The edges are round in mode 1 and square in mode 2.

A third, optional, argument can be added, to specify the object that gets the new setting, as shown in Figure TC.15 (top). Without a third argument, the setting applies to **all**.



(top) The radius for sticks was set for only the lower object, **pept**, by the command

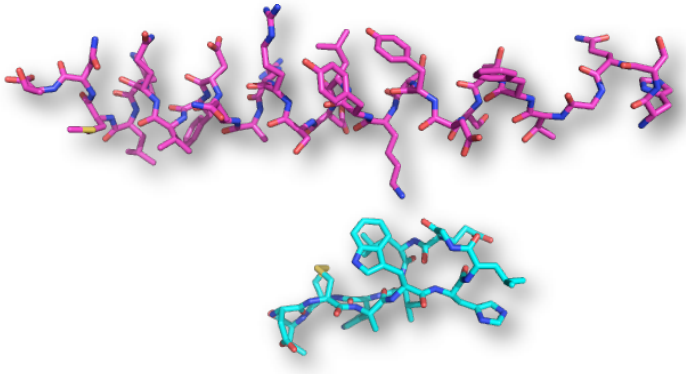
```
PyMOL> set stick_radius, 0.5, pept
```



(middle) The global value of stick_radius was then set to 0.0 by the command

```
PyMOL> unset stick_radius
```

The top object disappeared because its stick radius became 0.0.



(bottom) The `stick_radius` having been `set` for the object **pept**, the correct command for unsetting its `stick_radius` is

```
PyMOL> unset stick_radius, pept
```

Alternatively, the default value of `stick_radius`, 0.25, could be set for **all**

```
PyMOL> set stick_radius, 0.25, all
```

Figure TC.15

In general, settings are customizable for objects, including **all**, but not for selections (lists of atoms). However, there are exceptions: `sphere_color`, `surface_color`, `mesh_color`, `label_color`, `dot_color`, `cartoon_color`, `ribbon_color`, `transparency` (for surfaces), and `sphere_transparency` *can* be set for selections. While these settings can be altered for selections by typed commands, the current values of the settings for the given selections cannot be retrieved with the `get` command.

Like the `set` command, the `unset` command can apply globally, to specified objects, or to selections in certain cases. If you type an `unset` command with only the setting name argument, as in

```
PyMOL> unset stick_radius
```

the global value will be set to zero or to off. Figure TC.15 (middle) shows what may unexpectedly happen when the `stick_radius` is `set` for an object, but the `unset` command is given globally.

Command Line Features

Both PyMOL's upper and lower command lines provide shortcuts for typing commands and filenames.

Command completion using TAB

If you type the first few characters of a command and then hit TAB, PyMOL will either complete the command or print out a list of the commands that begin with the letters you typed. For example, if you type

```
PyMOL> sel
```

and then hit the TAB key, PyMOL will complete the command word,

```
PyMOL> select
```

If you type

```
PyMOL> se
```

and then hit TAB, PyMOL will provide a list of commands that begin with `se` in the command history and diagnostics area. If you hit the TAB key on a blank command line, PyMOL will provide a list of all its commands.

Command inference

If you are confident that the command you need begins with a unique string of a few letters, then you can type those few letters and PyMOL will infer the rest and perform the command. For example, rather than typing

```
PyMOL> select name ca
```

you can get the same result by typing

```
PyMOL> sel name ca
```

Command argument inference

The same principle works for command arguments. If the command argument you need begins with a unique string of a few letters, then you only need type those few letters. For example, rather than typing

```
PyMOL> show sticks, resn his
```

you can type

```
PyMOL> show st, resn his
```

and PyMOL recognize that `st` stands for the argument `sticks`. You can put command and argument completion together, and type

```
PyMOL> sh st, resn his
```

with the same result.

File name completion using TAB

If you use the command line for commanding input and output, rather than navigating through the file menu, you may have to enter some long paths and file names. PyMOL can increase your efficiency by completing unambiguous paths and file names. For instance, if a file named crystal.pdb exists in the current directory, typing

```
PyMOL> load cry
```

and hitting TAB will generate

```
PyMOL> load cystal.pdb
```

in the command line, and the file will be loaded when you hit return.

If there is some ambiguity in the file name, PyMOL will complete the name up to the point of ambiguity and then print (in the command history and diagnostics area) the matching file names in the directory.

Climbing the stack of commands

The PyMOL commands you type are kept accessible to you in a stack. PyMOL will display the previous command each time you press the up-arrow key on your keyboard. It displays the subsequent command when you press the down-arrow key. So if you want to repeat a command, use the arrow keys to display it, and then hit return.

Editing in the upper command line

The upper command line has an additional feature: it is editable. You can move the cursor in this (upper) command line using the left- and right-arrow keys; you can select, cut, copy and paste by dragging the cursor, and you can use the delete key to edit. You can copy and paste from one line of commands to another, and you can copy selection macros and commands from the command history area. On a Windows PC, select the text and use Ctrl-X to cut, Ctrl-C to copy, and Ctrl-V to paste. On a Mac, the apple key replaces Ctrl.

When you click on atoms in the Display Area, PyMOL provides a notice in the command history and diagnostics area identifying the atoms, as in

```
You clicked /1mbn//A/LYS`56/NZ
```

It's often convenient to cut an paste such macro identifiers into subsequent commands.

You can also cut and paste PyMOL commands to and from text files. This can be convenient when you are developing and testing a script.

Viewing commands in the Display Area

By default, command history and diagnostics are shown in the window above the upper command line. Alternatively, you can set PyMOL to show this text in the Display Area by choosing **Show Text**, which replaces the molecular display, or **Overlay Text**, which shows text over the molecular display, from the **Setting** menu. Choosing **Hide Text** will remove it from the Display Area.