```
                                        # are colored red.
```

Named selections will continue working after you have made changes to a molecular structure:

```
EXAMPLE

   PyMOL> load $PYMOL_PATH/test/dat/pept.pdb
   PyMOL> select bb, name c+o+n+ca     # The named selection "bb"
                                       # is created.

   PyMOL> count_atoms bb               # PyMOL counts 52 atoms in "bb."

   PyMOL> remove resi 5                # All atoms in residue 5 are removed
                                       # from the object "pept."

   PyMOL> count_atoms bb               # Now PyMOL counts
                                       # the remaining 48 atoms in "bb."
```

Named selections are static. Only atoms that exist at the time the selection is defined are included in the selection, even if atoms which are loaded subsequently fall within the selection criterion:

```
EXAMPLE

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

   PyMOL> select static_demo, pept    # The named selection "static_demo"
                                      # is created to reference all atoms.

   PyMOL> count_atoms static_demo     # PyMOL counts 107 atoms
                                      # in "static_demo."

   PyMOL> h_add                       # PyMol adds hydrogens in
                                      # the appropriate places

   PyMOL> count_atoms static_demo     # PyMOL still counts 107 atoms
                                      # in "static_demo,"
   PyMOL> count_atoms                 # even though it counts 200 atoms
                                      # in "pept."
```

Named selections can also be used in subsequent atom selections:

```
EXAMPLE

   PyMOL> select bb, name c+o+n+ca   # An atom selection named "bb"
                                     # is made, consisting of all
                                     # atoms named "C","O","N", or "CA."

   PyMOL> select c_beta_bb, bb or name cb
                                     # An atom selection named "c_beta_bb"
                                     # is made, consisting of
                                     # all atoms named "C", "O", "N", "CA" or "CB."
```

Note that the word "or" is used to select all atoms in the two groups, "bb" and "cb." The word "and" would have selected no atoms because it is interpreted in its boolean logical sense, not its natural language sense. See the subsection on "Selection Algebra" below.

Single−word Selectors

The very simplest *selection−expression*s are single−word selectors. These selectors do not take identifiers; they are complete by themselves.

Single Word Selector	Short Form Selector	Description
all	*	All atoms currently loaded into PyMOL
none	none	No atoms (empty selection)
hydro	h.	All hydrogen atoms currently loaded into PyMOL
hetatm	het	All atoms loaded from Protein Data Bank HETATM records
visible	v.	All atoms in enabled objects with at least one visible representation
present	pr.	All atoms with defined coordinates in the current state (used in creating movies)

The selector **none** won't come up much when you are typing commands directly into PyMOL, but it is useful in programming scripts.

As the table shows, many single−word selectors have short forms to save on typing. Some short forms must be followed by a period and a space, in order to delimit the word. Short forms and long forms have the same effect, so choose the form that suits you:

```
EXAMPLES

   PyMOL> color blue, all          # It all turns blue.
   PyMOL> color blue, *

   PyMOL> hide hydro               # Representations of all
   PyMOL> hide h.                  # hydrogen atoms are hidden.

   PyMOL> show spheres, hetatom    # All the atoms defined as HETATOMS
   PyMOL> show spheres, het        # in the PDB input file
                                   # are represented as spheres.
```

Property Selectors

PyMOL reads data files written in PDB, MOL/SDF, Macromodel, ChemPy Model, and Tinker XYZ formats. Some of the data fields in these formats allow PyMOL to assign properties to atoms. You can group and select atoms according to these properties using property selectors and identifiers: the selectors correspond to the fields in the data files, and the identifiers correspond to the target words to match, or the target numbers to compare.

The items in a list of identifiers are separated by plus signs (+) only. Do not add spaces within a list of identifiers. The selector **resi** takes (+)−separated lists of identifiers, as in

```
EXAMPLE

   PyMOL> select nterm, resi 1+2+3
```

or, alternatively, it may take a range given with a dash:

```
EXAMPLE
```

```
   PyMOL> select nterm, resi 1-3
```

However, you will get an error message if you try to combine a list and a range in an identifier to a **resi** as in "select mistake, resi 1−3+6."

The identifier for a blank field in an input file is and empty pair of quotes:

```
EXAMPLE

   PyMOL> select unstruct, ss ""   # A named selection is created
                                   # to contain all atoms that are not assigned
                                   # a secondary structure.
```

Most *property selector*s select matches to their identifiers:

Matching Property Selector	**Short Form Selector**	**Identifier and Example**
symbol	e.	*chemical−symbol−list* list of 1− or 2−letter chemical symbols from the periodic table `PyMOL> select polar, symbol o+n`
name	n.	*atom−name−list* list of up to 4−letter codes for atoms in proteins or nucleic acids `PyMOL> select carbons, name ca+cb+cg+cd`

resn	r.	*residue−name−list*
		list of 3−letter codes for amino acids
		
		`PyMOL> select aas, resn asp+glu+asn+gln`
		
		or list of up to 2−letter codes for nucleic acids
		
		`PyMOL> select bases, resn a+g`

resi	i.	*residue−identifier−list*
		list of up to 4−digit residue numbers
		
		`PyMOL> select mults10, resi 1+10+100+1000`
		
		residue−identifier−range
		
		`PyMOL> select nterm, resi 1-10`

alt	alt	*alternate−conformation−identifier−list* list of single letters `PyMOL> select altconf, alt a+""`
chain	c.	*chain−identifier−list* list of single letters or sometimes numbers

		`PyMOL> select firstch, chain a`
segi	s.	*segment−identifier−list* list of up to 4 letter identifiers `PyMOL> select ligand, segi lig`
flag	f.	*flag−number* a single integer from 0 to 31 `PyMOL> select f1, flag 0`
numeric_type	nt.	*type−number* a single integer `PyMOL> select type1, nt. 5`
text_type	tt.	*type−string* a list of up to 4 letter codes `PyMOL> select subset, text_type HA+HC`
id	id	*external−index−number* a single integer `PyMOL> select idno, id 23`
index	idx.	*internal−index−number* a single integer `PyMOL> select intid, index 11`
ss	ss	*secondary−structure−type* list of single letters `PyMOL> select allstrs, ss h+s+l+""`

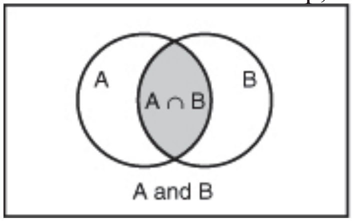
Other *property selector*s select by comparison to numeric identifiers:

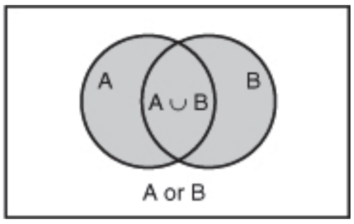
Numeric Selector	Short Form	Argument &Example
b	b	*comparison−operator b−factor−value* a real number `PyMOL> select fuzzy, b > 10`
q	q	*comparison−operator occupancy−value* a real number `PyMOL> select lowcharges, q <0.50`
formal_charge	fc.	*comparison−operator formal charge−value* an integer `PyMOL> select doubles, fc. = −1`
partial_charge	pc.	*comparison−operator partial charge−value* a real number `PyMOL> select hicharges, pc. > 1`

Details of the atom and residue name formats can be found in the official guide to PDB file formats, http://www.rcsb.org/pdb/docs/format/pdbguide2.2/guide2.2_frame.html.

Selection Algebra

Selections can be made more precise or inclusive by combining them with logical operators, including the boolean **and, or** and **not**. The boolean **and** selects only those items that have both (or all) of the named properties, and the boolean **or** selects items that have either (or any) of them. Venn diagrams show that **and** selects the areas of overlap, while **or** selects both areas.





Operators:

Selection operators and modifiers are listed below. The dummy variables *s*1 and *s*2 stand for selection−expressions such as "chain a" or "hydro."

Operator	**Short form**	**Effect**
not *s*1	! *s*1	Selects atoms that are not included in *s*1 `PyMOL> select sidechains, ! bb`
*s*1 and *s*2	*s*1 & *s*2	Selects atoms included in both *s*1 and *s*2 `PyMOL> select far_bb, bb &farfrm_ten`
*s*1 or *s*2	*s*1 | *s*2	Selects atoms included in either *s*1 or *s*2 `PyMOL> select all_prot, bb | sidechain`
*s*1 in *s*2	*s*1 in *s*2	Selects atoms in *s*1 whose identifiers name, resi, resn, chain and segi all match atoms in *s*2 `PyMOL> select same_atms, pept in prot`
*s*1 like *s*2	*s*1 l. *s*2	Selects atoms in *s*1 whose identifiers name and resi match atoms in *s*2 `PyMOL> select similar_atms, pept like prot`
*s*1 gap *X*	*s*1 gap *X*	Selects all atoms whose van der Waals radii are separated from the van der Waals radii of *s*1 by a minimum of *X* Angstroms. `PyMOL> select farfrm_ten, resi 10 gap 5`
*s*1 around *X*	*s*1 a. *X*	Selects atoms with centers within *X* Angstroms of the center of any atom in *s*1 `PyMOL> select near_ten, resi 10 around 5`
*s*1 expand *X*	*s*1 e. *X*	Expands *s*1 by all atoms within *X* Angstroms of the center of any atom in *s*1 `PyMOL> select near_ten_x, near10 expand 3`

*s*1 within X of *s*2	*s*1 w. X of *s*2	Selects atoms in *s*1 that are within X Angstroms of the *s*2 `PyMOL> select bbnearten, bb w. 4 of resi 10`
byres *s*1	br. *s*1	Expands selection to complete residues `PyMOL> select complete_res, br. bbnear10`
byobject *s*1	bo. *s*1	Expands selection to complete objects `PyMOL> select near_obj, bo. near_res`
neighbor *s*1	nbr. *s*1	Selects atoms directly bonded to *s*1 `PyMOL> select vicinos, neighbor resi 10`

Logical selections can be combined. For example, you might select atoms that are part of chain a, but not residue number 125:

```
EXAMPLE

   PyMOL> select chain a and (not resi 125)      # selects atoms that are part of
                                                 # chain a, but not
                                                 # residue number 125.

   PyMOL> select (name cb or name cg1 or name cg2) and chain A    # These two
                                                                  # selections are
   PyMOL> select name cb+cg1+cg2 and chain A                      # equivalent.
                                                                  # select c-beta's,
                                                                  # c-gamma-1's and
                                                                  # c-gamma-2's
                                                                  # that are
                                                                  # in chain A.
```

Like the results of groups of arithmetic operations, the results of groups of logical operations depend on which operation is performed first. They have an order of precedence. To ensure that the operations are performed in the order you have in mind, use parentheses:

```
   byres ((chain a or (chain b and (not resi 125))) around 5)
```

PyMOL will expand its logical selection out from the innermost parentheses.

Atom Selection Macros

Macros make it possible to represent a long atom selection phrase such as

```
PyMOL> select pept and segi lig and chain b and resi 142 and name ca
```

in a more compact form:

```
PyMOL> select /pept/lig/b/142/ca
```

An atom selection macro uses slashes to define fields corresponding to identifiers. The macro is used to select atoms using the boolean "and," that is, the selected atoms must have all the matching identifiers:

```
   /object-name/segi-identifier/chain-identifier/resi-identifier/name-identifier
```

These identifiers form a hierarchy from the object−name at the top, down to the name−identifier at the bottom. PyMOL has to be able to recognize the macro as one word, so no spaces are allowed within it.

Macros come in two flavors: those that begin with a slash and those that don't. The presence or absence of a slash at the beginning of the macro determines how it is interpreted. If the macro begins with a slash, PyMOL expects to find the fields *starting* from the *top* of the hierarchy: the first field to the right of the slash is interpreted as an object−name; the second field as an identifier to segi; the third as an identifier to chain, and so on. It may take any of the following forms:

```
   /object-name/segi-identifier/chain-identifier/resi-identifier/name-identifier
   /object-name/segi-identifier/chain-identifier/resi-identifier
   /object-name/segi-identifier/chain-identifier
   /object-name/segi-identifier
   /object-name
```

```
EXAMPLES

   PyMOL> zoom /pept
   PyMOL> show spheres, /pept/lig/
   PyMOL> show cartoon, /pept/lig/a
   PyMOL> color pink, /pept/lig/a/10
   PyMOL> color yellow, /pept/lig/a/10/ca
```

If the macro does not begin with a slash, it is interpreted differently. In this case, PyMOL expects to find the fields *ending* with the *bottom* of the hierarchy. Macros that don't start with a slash may take the following forms:

```
                                                 resi-identifier/name-identifier
                                chain-identifier/resi-identifier/name-identifier
            segi-identifier/chain-identifier/resi-identifier/name-identifier
object-name/segi-identifier/chain-identifier/resi-identifier/name-identifier
```

```
EXAMPLES

   PyMOL> zoom 10/cb
   PyMOL> show spheres, a/10-12/ca
   PyMOL> show cartoon, lig/b/6+8/c+o
   PyMOL> color pink, pept/enz/c/3/n
```

You can also omit fields between slashes. Omitted fields will be interpreted as wildcards, as in the following forms:

```
   resi-identifier/
   resi-identifier/name-identifier
   chain-identifier//
   object-name//chain-identifier
```

```
EXAMPLES

   PyMOL> zoom 142/                  # Residue 142 fills the viewer.

   PyMOL> show spheres, 156/ca       # The alpha carbon of residue 156
                                     # is shown as a sphere

   PyMOL> show cartoon, a//          # Chain "A" is shown as a cartoon.

   PyMOL> color pink, pept//b        # Chain "B" in object "pept"
                                     # is colored pink.
```

Selection macros must contain at least one forward slash in order to distinguish them from other words in the selection language. Being words, they must not contain any spaces. When using macros, it is also important to understand that they are converted into long form before being submitted to the selection engine. This can help in the interpretation of error messages.

Calling Python from within PyMOL

Single−line Python statements can be issued directly within PyMOL. For example:

```
PyMOL> print 1 + 2
3
```

Full access is available to standard Python library functions, and you can assign results to symbols.

```
PyMOL>import time
PyMOL>now = time.time()
PyMOL>print now
1052982734.94
```

Multi−line blocks of Python can be included within PyMOL command scripts provided that a backslash ('\') is used for continuation on all but the final line.

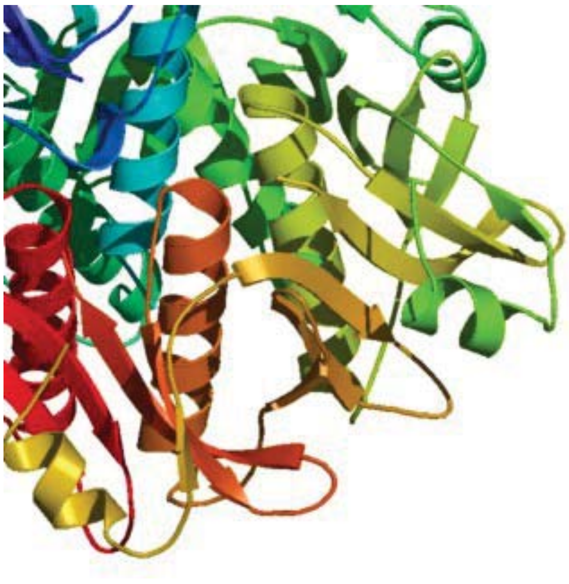
```
PyMOL> for a in range(1,6): \
PyMOL>    b = 6 - a \
PyMOL>    print a, b
1 5
2 4
3 3
4 2
5 1
```

Cartoon Representations

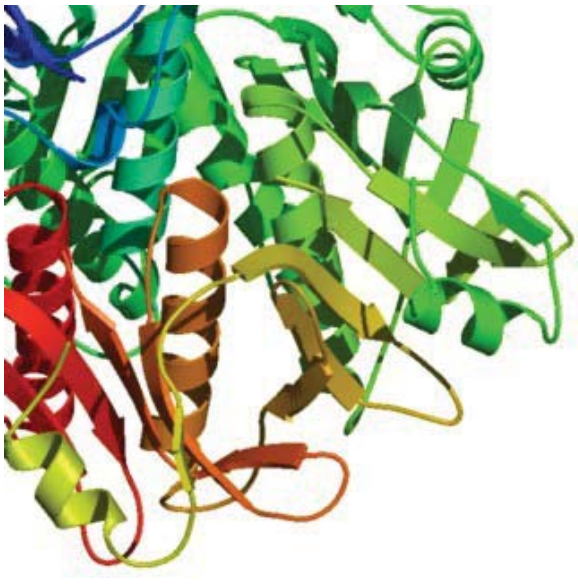
Background

Accessibility

Cartoon ribbons in PyMOL rival those of the popular Molscript/Raster3D packages, but PyMOL makes creating high quality images much easier. While PyMOL can read Molscript output directly (see the chapter on Molscript), this is no longer necessary or as convenient as utilizing PyMOL's built−in cartoon ribbon capability:



PyMOL built−in ribbons



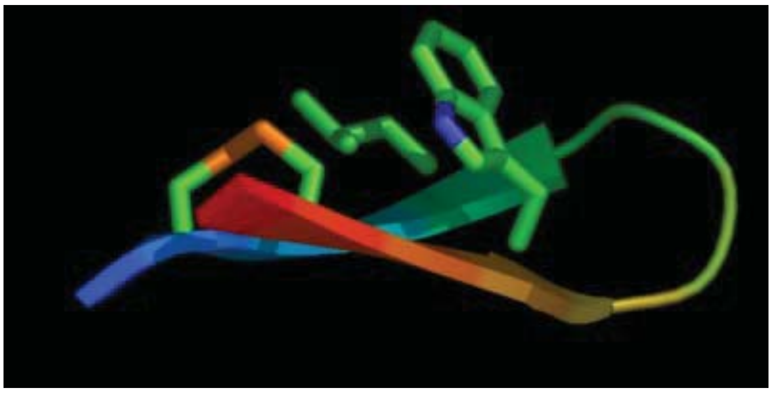
"molauto −nice ... | molscript −r > ..."

Molscript's cartoons are slightly more ideal, but PyMOL comes pretty darn close!

Note that all of the images in this section were colored using the rainbow feature (Color pop−up menu) and ray−traced with antialising enabled.

Pretty *and* Correct

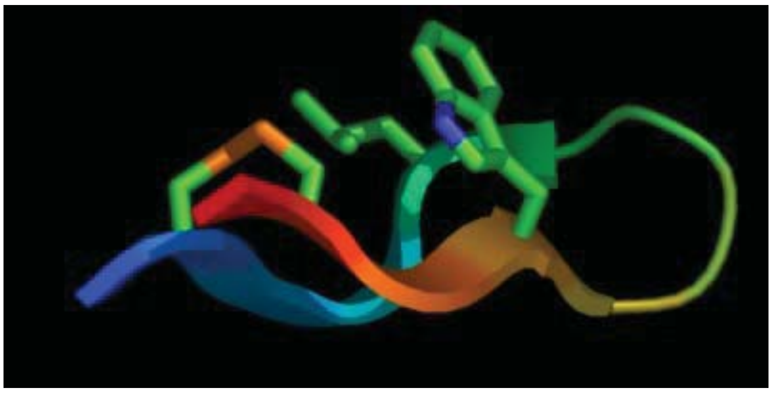
One of the advantages of PyMOL's cartoon ribbon facility is that it is easy to switch between "smoothed" versions of protein secondary structure, and "correct" renditions which portray actual main chain coordinates. Although cartoons are often used solely to represent protein structures in a schematic sense, sometimes it is desirable to combine a schematic overall picture with atomic resolution in one particular location. However, unless the cartoon track properly with alpha−carbon positions, the resulting figures will look a little silly:



In the above image, the side chains are floating off into space. Disabling "flat sheets" from the Cartoons Menu or issuing the command

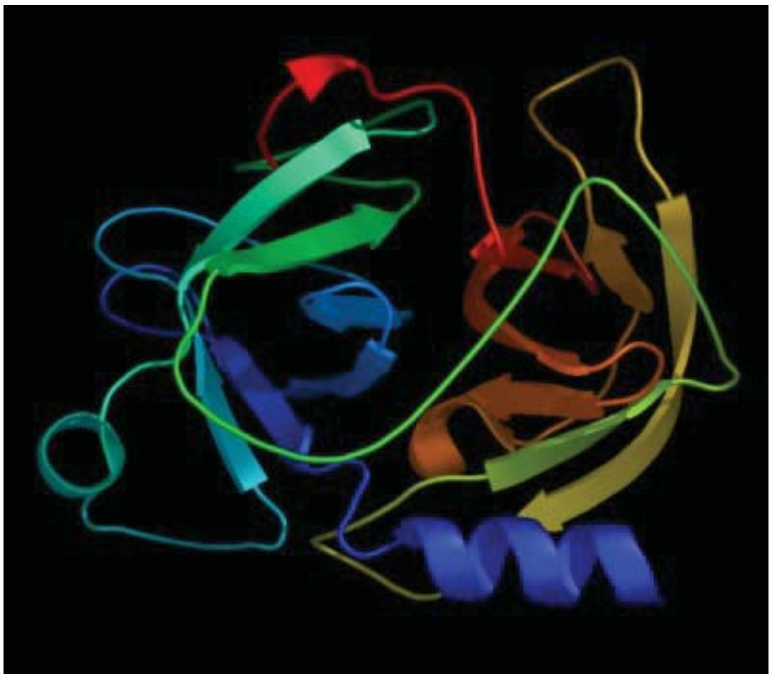
```
set cartoon_flat_sheets, 0
```

will make the beta strands follow the true path of the backbone through space and give a more accurate rendition of the structure.



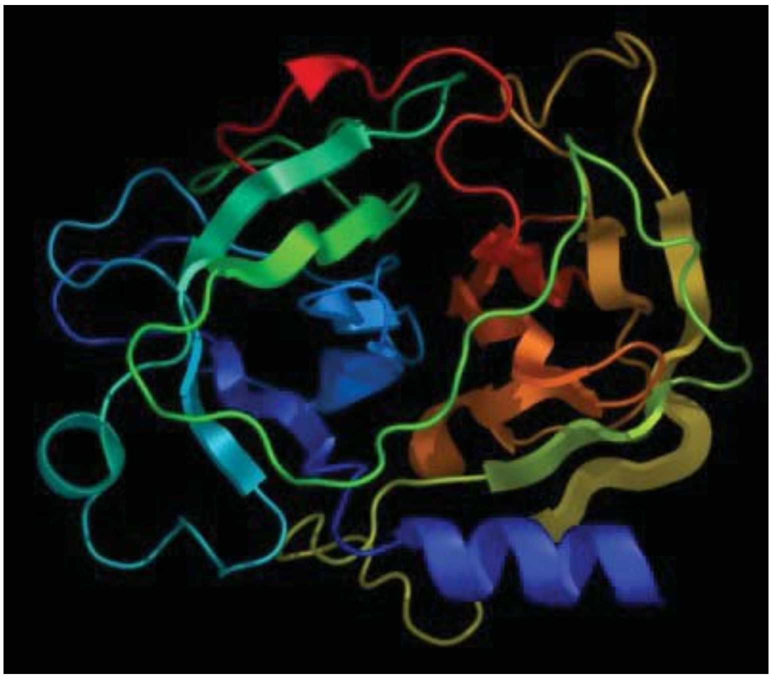
The appearance of a cartoon over the entire molecule will be substantially different when all smoothing features are turned off. For instance, with smoothing enabled:

```
set cartoon_flat_sheets, 1
set cartoon_smooth_loops, 1
```

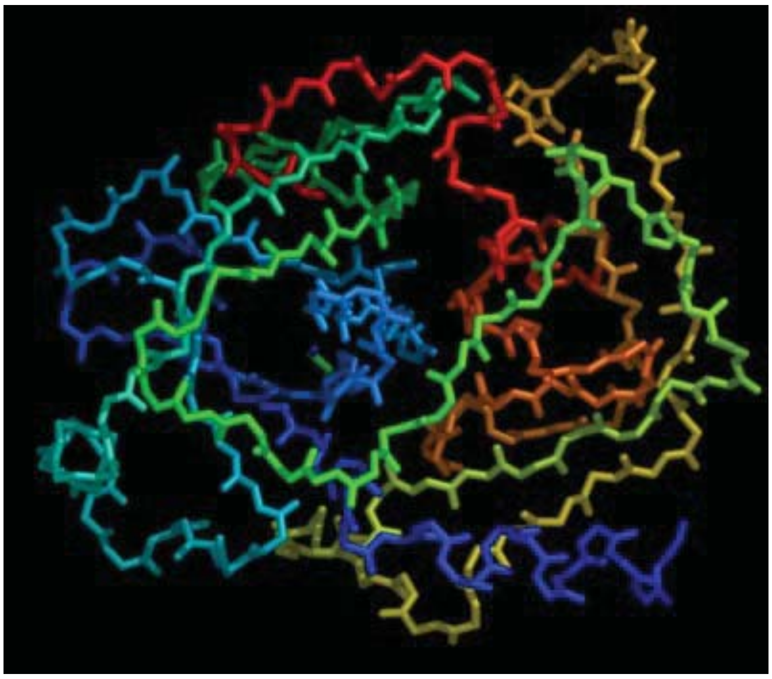


the image differs substantially from:

```
set cartoon_flat_sheets, 0
set cartoon_smooth_loops, 0
```



which more accurately reflects the true path of the peptide backbone:



To facilitate beautiful imagery, smoothing is enabled by default (just like Molscript) [NOTE: THIS MAY CHANGE BEFORE VERSION 1.0] . Just be sure to turn it off when you want to study structures at atomic resolution (*remember, real life is a bit more complicated than what you see in cartoons*!).

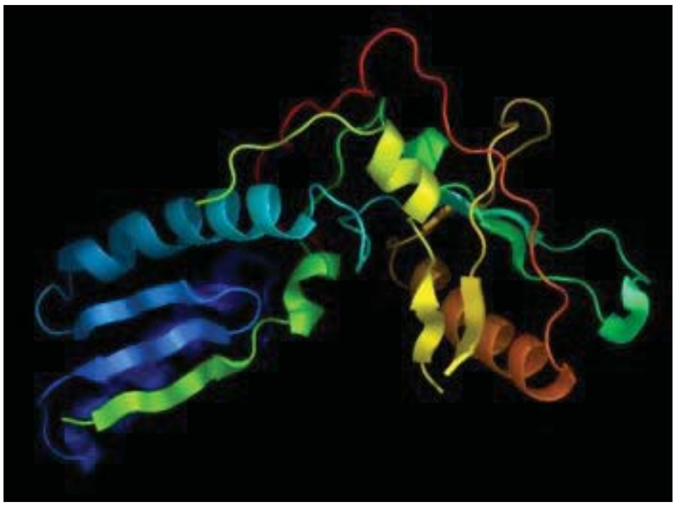
Customization

Cartoon Types

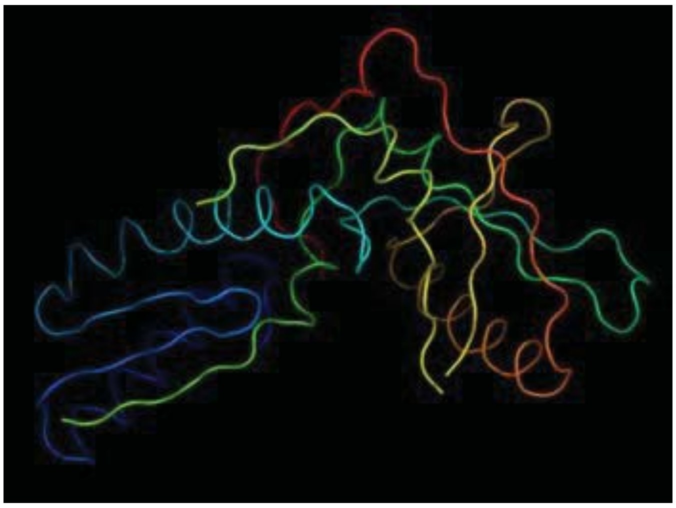
Best results will be obtained when secondary structure information has been defined for each residue in the molecule. Under these conditions, PyMOL will do extra processing to insure that good normals have been calculated for helical regions, and perform smoothing of loops, where desired.

Also under such conditions, in automatic mode, cartoon representations will be assigned according to the secondary structure type. However, you can instruct PyMOL to ignore such information, and manually control when and where various cartoon representations are employed.

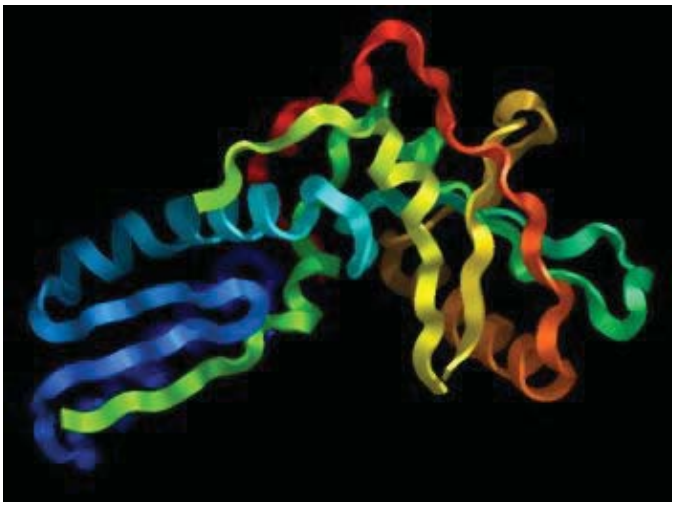
```
show cartoon
cartoon automatic       # default
```



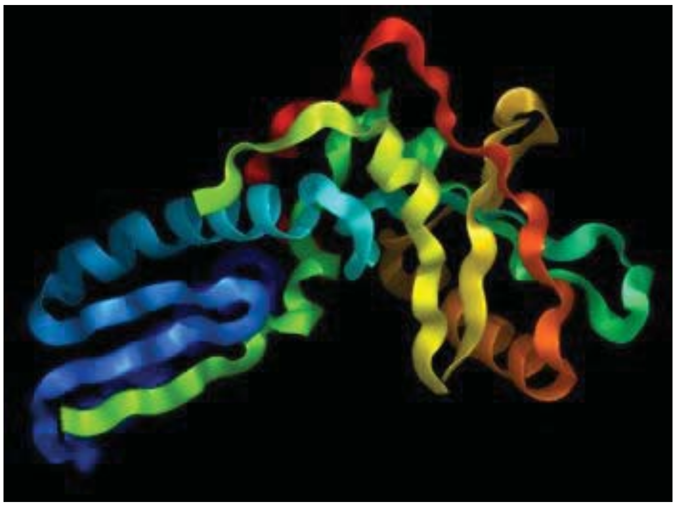
`cartoon loop`



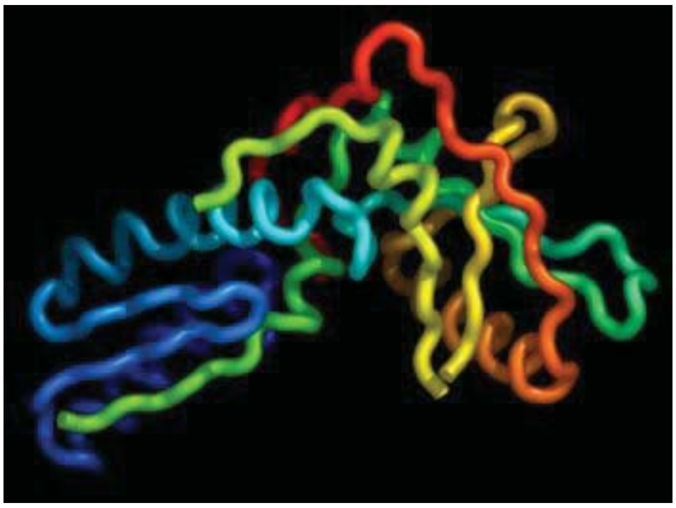
`cartoon rect`



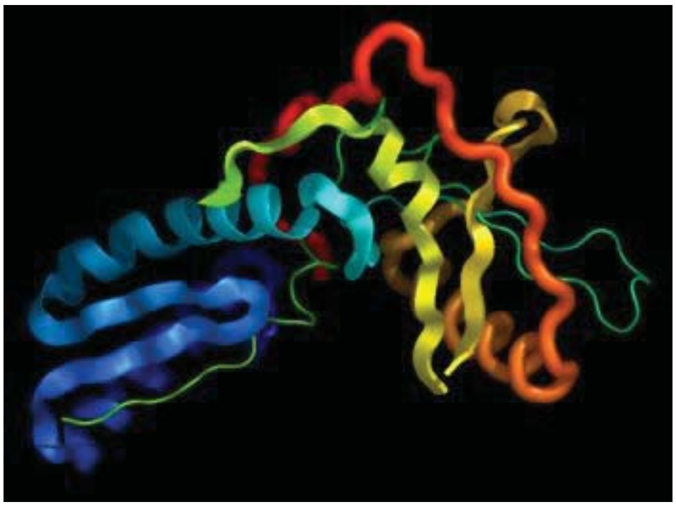
`cartoon oval`



```
cartoon tube
```



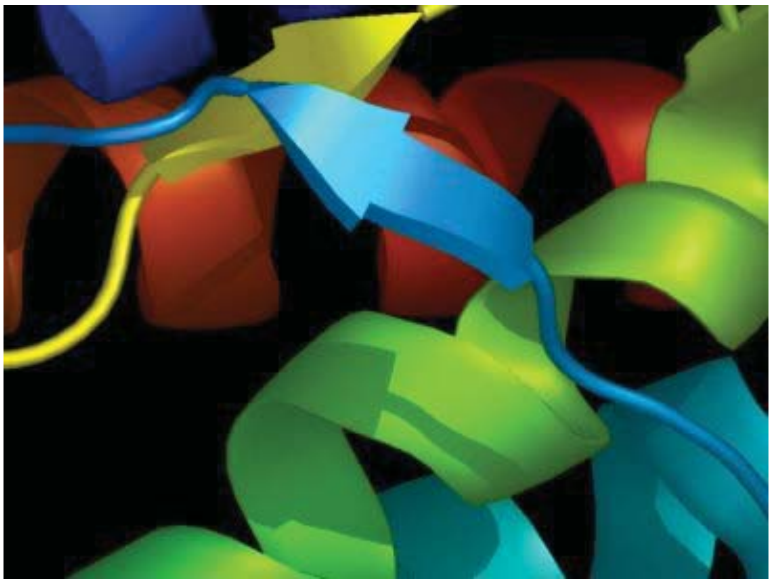
```
cartoon tube, 1:49/
cartoon arrow, 50:99/
cartoon loop, 100:149/
cartoon oval, 150:199/
cartoon rect, 200:250/
```



All cartoon ribbons have associated parameters accessible from the "set" command which allow you to change their appearance. See the chapter on Settings for more information.

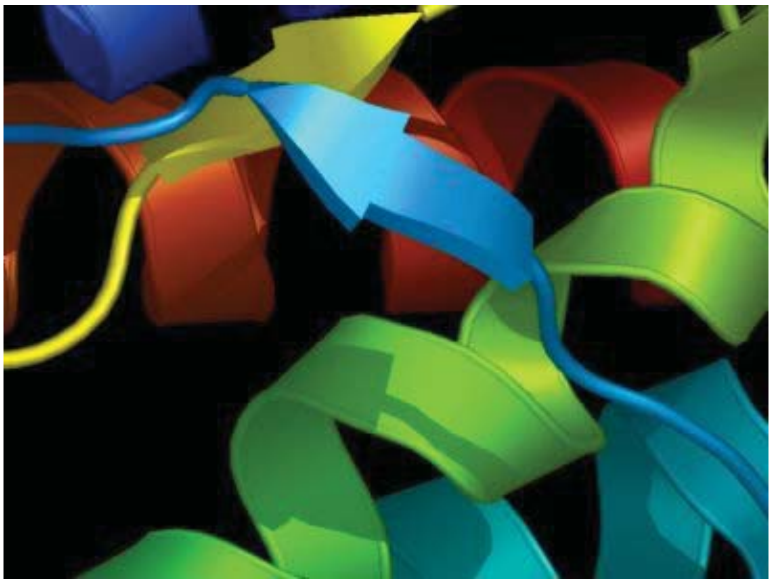
Fancy Helices

```
set cartoon_fancy_helices, 0
```



Molscript addicts who simply must have those ribbon helices with tubular edges will not be disappointed with "fancy helices":

```
set cartoon_fancy_helices, 1
```



Secondary Structure Assignment

It is recommended that you read in PDB files which already have correct secondary structure assignments from a program like DSSP. However, PyMOL does have a reasonably fast secondary structure alignment algorithm called "dss". Please be aware that due to the subjective nature of secondary structure assignment in borderline cases, dss results will differ somewhat from DSSP.

```
SYNTAX

   dss selection

EXAMPLE

   dss 1dfr
```

If you are visualizing an animation, you may wish derive secondary structure assignment from a specific state of the animation. This can be done with:

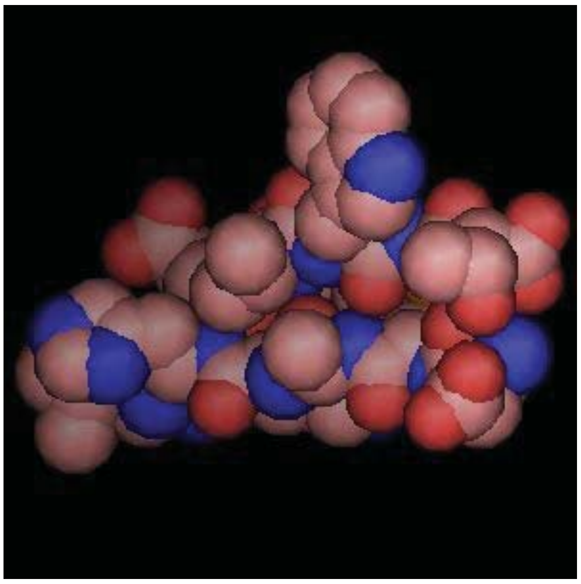
```
SYNTAX

   dss selection, state

EXAMPLE

   dss mov, 1
```

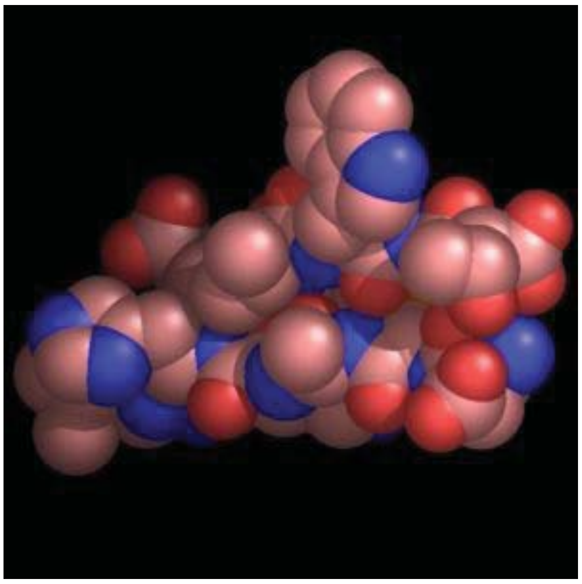
To change assignments manually, the best way is to use the alter command as follows:

```
show cartoon
alter 11-40/, ss='H'            # assign residues 11-40 as helix
alter 40-52/, ss='L'            # assign residues 40-52 as loop
alter 52-65/, ss='S'            # assign residues 52-65 as sheet
alter 65-72/, ss='H'            # assign residues 65-72 as helix
rebuild                         # regenerate the cartoon
```

Ray−Tracing

Ray−tracing produces the highest quality molecular graphics images. **PyMOL is the first full−featured molecular graphics program to include a high−speed ray−tracer** which works with its native internal geometries (except text).


OpenGL Rendering (real−time manipulation)


Ray−traced Rendering (seconds or minutes per frame)

You can ray−trace any Scene in PyMOL by clicking the "Ray Trace" button in the external GUI or using the "ray" command. The built−in raytracer also makes it possible to easily assemble very high−quality movies in a snap.

Important Settings

These can be changed using the "set" command. Unless otherwise specified, the settings apply only to the ray−tracing engine and not the OpenGL renderer. Some reconciliation between the two renderers is much needed, so be warned that these settings may change in the future.

Normally, the only settings you will need to change are **orthoscopic**, **antialias**, and **gamma**. If you are down in an enzyme active site which is heavily shadowed, you may want to increase **direct** to 0.5−0.7 in order to improve brightness and contrast.

- **orthoscopic** (0 or 1) controls whether the OpenGL renderer uses the same orthoscopic transformation as the renderer. You'll want to set this to 1 when preparing figures so that OpenGL and raytracing match pixel−for−pixel.
- **ambient** (0.0−1.0) controls the ambient light intensity for both OpenGL and the ray−tracer.
- **ambient_scale** (float) controls the relative ambient intensity between OpenGL and the ray−tracer.
- **antialias** (0 or 1) generate a "smooth" image (best quality, but takes 4X as long).
- **direct** (0.0−1.0) the planer light intesity originating from the camera.