

## Appendix A: Command Reference

Python Commands and Syntax	
<pre># I am a comment; Python ignores me. """I am also ignored. Me too! """</pre>	Comments Block comments (doc strings)
<pre>i = 1 # (After the # is ignored.) first_name = "Bob" j = 1.0 i_am_a_boolean = True i_am_an_integer = not i_am_a_boolean k = first_name</pre>	Simple variable assignments (Python is case sensitive.)
<pre>tuple = (1, 2, 3) list = [15, 'X', -1.5, "lion"] 2_D_list[[1, 0], [0, 1]] dict = {"apple": "sour", "days": 24}</pre>	Assignment to various sequences (tuples, lists, and dictionaries)
<pre>print i + 2, 3/2, float(3)/2,       3.0/2.0, 2*j, (i + 3)**2</pre>	Outputs 3 1 1.5 1.5 2.0 16 to the screen. (Python returns the floor of integer-only calculations, so convert at least one to a float if needed.)
<pre>print k + " thinks " + str(i) +       " = 0."</pre>	Outputs Bob thinks 1 = 0. (Python will not concatenate objects of different types, so a function must be used to convert an integer to a string.)
<pre>print list[1], tuple[1:2], k[0:2],       dict["days"],       2_d_list[0][1]</pre>	Outputs 15 (2, 3) Bo 24 0. (Lists, strings, etc., are indexed starting with 0.)
<pre>i += 1 j -= 1 print i, j</pre>	Increment and decrement operators. Outputs 2 0.0
<pre>tuple[2] = 4 first_name[0] = 'R'</pre>	Raise errors; tuples and strings are immutable.
<pre>list[3] = "tiger" dict["apple"] = "sweet"</pre>	Lists and dictionaries are mutable.
<pre>list.append(False) dict["new entry"] = 'Z' print list, dict</pre>	Outputs [15, 'X', -1.5, "tiger", False] {"apple": "sweet", "days": 24, "new entry", 'Z'}
<pre>for i in range(1, 10):     print i</pre>	The newly defined variable <i>i</i> ranges from 1 up to <i>but not including</i> 10, and the command <code>print i</code> is executed for each value.
<pre>for j in ("cats", "dogs", "fish"):     print j</pre>	Outputs: <b>cats</b> <b>dogs</b> <b>fish</b>
<pre>if x &lt; 0:     print "negative" elif x == 0:     print "zero" else:     print "positive"</pre>	Conditional statement that executes lines only if Boolean statements are true. <code>elif</code> means "or <u>e</u> lse, check <u>i</u> f". Do not mix up = and ==! Use indenting to indicate blocks of code executed together under the conditional.

<pre>if i_am_a_boolean:     print "Yes!" if not i_am_an_integer:     print "No!"</pre>	<p>Outputs: <b>Yes!</b> <b>No!</b></p>
<pre>a, b = 1, 1 print a is b print a == b</pre>	<p>Outputs: <b>False</b> <b>True</b> (a and b contain the same value, but are not two names for the same object!)</p>
<pre>while len(list) &lt;= 7:     list.append("blah") print list</pre>	<p>Outputs: [15, 'X', -1.5, "tiger", False, "blah", "blah", "blah"]</p>
<pre>def my_func(a, b):     c = a + b     d = b + c     e = c + d     return c, d, e</pre>	<p>Defines a function (returns a value) or subroutine (does not). If a and b were 1 and 2, this function would return (3, 5, 8)</p>
<pre>returned_values = my_func(1, 2) value_of_c = returned_values[0] value_of_d = returned_values[1] value_of_e = returned_values[2]</pre>	<p>Syntax for using multiple values returned by a function, e.g., value_of_c is 3.</p>
<pre>class MyCircle:     """This is a MyCircle class."""     def __init__():         #Code to run when a Circle         object is instantiated         goes here.         self.radius = 1.0 # Sets         default value for radius      def draw(self, color=0):         #Code to draw the circle         goes here.         Pass</pre>	<p>Defines a class with two methods</p>
<pre>circle = MyCircle() circle.draw() circle.radius = 1.5 circle.draw(1)</pre>	<p>Constructs a MyCircle object, draws a circle in color 0, and then draws a circle in color 1</p>
<pre>file = open("out.txt", 'w') file.write("hello") file.close()</pre>	<p>Opens a new file named out.txt for <u>w</u>riting and outputs hello to the file. (Be sure to close your files when finished with them.)</p>
<pre>import module</pre>	<p>Imports and runs the module module.py so that its functions can be called with module.function().</p>
<pre>from module import function</pre>	<p>Imports the specific function function so that it can be called with simply function().</p>
<pre>from module import *</pre>	<p>Imports all of the (public) functions from module.py.</p>

<b>Python Math</b>	
<code>math.exp(5)</code>	Returns the value of $e^5$
<code>math.pi</code>	Returns the value of $\pi$
<code>math.sin(theta)</code>	Returns the value of $\sin \theta$ , where $\theta$ is in radians
<code>math.acos(x)</code>	Returns the value of $\arccos x$ in radians
<code>math.degrees(rad)</code>	Converts radians to degrees
<code>math.radians(deg)</code>	Converts degrees to radians
<code>random.random()</code>	Returns a random floating point number between 0.0 and 1.0
<code>random.randint(5, 10)</code>	Returns a random integer between 5 and 10 (inclusive)
<code>random.gauss(10, 2)</code>	Returns a random number from a Gaussian distribution with a mean of 10 and a standard deviation of 2

<b>Rosetta: Vector Calculus</b>	
<code>v = numeric.xyzVector_float(x, y, z)</code>	Creates a displacement vector with floating point precision from Cartesian coordinates
<code>print v</code>	Outputs <code>v</code> and its elements
<code>print v.x, v.y, v.z</code>	
<code>v - v2</code>	Returns the displacement vector between <code>v</code> and <code>v2</code>
<code>v.norm</code>	Returns the vector norm of <code>v</code>
<code>v.dot(v2)</code>	Returns the dot product of <code>v</code> and <code>v2</code>
<code>v.cross(v2)</code>	Returns the cross product of <code>v</code> and <code>v2</code>

<b>Rosetta: Toolbox Methods</b>	
<code>cleanATOM("1YY8.pdb")</code>	Creates a "cleaned" pdb file with all non-ATOM lines of a pdb file removed
<code>cleanCRYS("1YY8.pdb", 2)</code>	Creates a "cleaned" crystal structure that removes redundant crystal contacts and isolates a monomer.
<code>pose = pose_from_rcsb("1YY8")</code>	Loads pdb 1YY8 from the Internet
<code>generate_resfile_from_pdb("input.pdb", "output.resfile")</code>	Generate a resfile from a pdb file or a pose, respectively
<code>generate_resfile_from_pose(pose, "output.resfile")</code>	
<code>mutate_residue(pose, 49, 'E')</code>	Replaces residue 49 of pose with a glutamate (E) residue (does not optimize rotamers)
<code>get_secstruct(pose)</code>	Assigns secondary structure information to pose and outputs it to the screen
<code>hbond_set = get_hbonds(pose)</code>	Instantiates and fills an H-bond set with hydrogen-bonding data

<b>Rosetta: Pose Object</b>	
<code>pose = Pose()</code>	Instantiates an empty <code>pose</code> object from the <code>Pose</code> class
<code>pose = pose_from_pdb("input_file.pdb")</code>	Loads a <code>pdb</code> file from the working directory into a new <code>pose</code> object
<code>pose = pose_from_rcsb("1YY8")</code>	Loads <code>pdb</code> 1YY8 from the Internet
<code>pose = pose_from_sequence("AAAAAA", "fa_standard")</code>	Creates a new <code>pose</code> from the given sequence using standard, full-atom residue type templates
<code>print pose</code>	Displays information about the <code>pose</code> object: <code>pdb</code> filename, sequence, and fold tree
<code>pose.sequence()</code>	Returns the sequence of the <code>pose</code> structure
<code>get_secstruct(pose)</code>	Assigns secondary structure information to <code>pose</code> and outputs it to the screen
<code>pose.assign(other_pose)</code>	Copies <code>other_pose</code> onto <code>pose</code> . You cannot simply type <code>pose = other_pose</code> , as that will only point <code>pose</code> to <code>other_pose</code> and not actually copy it.
<code>pose.dump_pdb("output_file.pdb")</code>	Creates a <code>pdb</code> file named <code>output_file.pdb</code> in the working directory using information from <code>pose</code> object.
<code>pose.is_fullatom()</code>	Returns <code>True</code> if the <code>pose</code> contains a full-atom representation of a structure
<code>pose.total_residue()</code>	Returns total number of residues in the <code>pose</code>
<code>pose.phi(5)</code> <code>pose.psi(5)</code> <code>pose.omega(5)</code> <code>pose.chi(2, 5)</code>	Returns the $\phi$ , $\psi$ , $\omega$ or $\chi_2$ angles (in degrees) of the 5 <sup>th</sup> residue in the <code>pose</code>
<code>pose.set_phi(5, 60.0)</code> <code>pose.set_psi(5, 60.0)</code> <code>pose.set_omega(5, 60.0)</code> <code>pose.set_chi(2, 5, 60.0)</code>	Sets the $\phi$ , $\psi$ , $\omega$ or $\chi_2$ angles of the 5 <sup>th</sup> residue in the <code>pose</code> to 60.0°
<code>print pose.residue(5)</code>	Outputs the amino acid details of residue 5
<code>pose.residue(5).name()</code>	Returns the 3-letter residue name for residue 5
<code>pose.residue(5).is_polar()</code>	Return <code>True</code> if the 5 <sup>th</sup> residue is of the queried type
<code>pose.residue(5).is_aromatic()</code>	
<code>pose.residue(5).is_charged()</code>	
<code>pose.residue(5).xyz("CA")</code>	Return the displacement vector of the $\alpha$ carbon (CA) of residue 5, which is the 2 <sup>nd</sup> atom listed for that residue in the <code>pose</code> and a standard <code>pdb</code> file
<code>pose.residue(5).xyz(2)</code>	Returns 2
<code>pose.residue(5).atom_index("CA")</code>	Returns 2
<code>for i in range(1, pose.total_residue() + 1):     print pose.residue(i).name()</code>	Loops through all residues in the <code>pose</code> and outputs the 3-letter name of each (Unlike Python, Rosetta indexes residues starting with 1.)
<code>atom = pose.residue(5).atom("CA")</code>	Constructs an atom object for the $\alpha$ carbon (CA) of residue 5
<code>atom1 = AtomID(1, 5)</code> <code>atom2 = AtomID(2, 5)</code> <code>atom3 = AtomID(3, 5)</code>	Construct unique atom <i>identifier</i> objects for the 1 <sup>st</sup> , 2 <sup>nd</sup> , and 3 <sup>rd</sup> , atoms of <i>any</i> residue 5, respectively (This is not the same as the above command!)

<code>pose.conformation().bond_length(atom1, atom2)</code>	Returns the bond length (if stored in the conformation object) between atom1 and atom2
<code>pose.conformation().bond_angle(atom1, atom2, atom3)</code>	Returns the bond angle in radians (if stored in the conformation object) of atom1, atom2, and atom3
<code>pose.conformation().set_bond_length(atom1, atom2, 1.5)</code>	Sets the bond length between atom1 and atom2 to 1.5 Å
<code>pose.conformation().set_bond_angle(atom1, atom2, atom3, 0.66666 * math.pi)</code>	Sets the bond angle of atom1, atom2, and atom3 to ~120°
<code>print pose.pdb_info()</code>	Displays a table comparing the sequence numbering range in the pose with that of the pdb file from which the pose was generated
<code>pose.pdb_info().name()</code>	Returns the filename of the pdb file from which the pose was generated
<code>pose.pdb_info().number(5)</code>	Returns the pdb number of pose residue 5
<code>pose.pdb_info().chain(5)</code>	Returns the pdb chain label of pose residue 5
<code>pose.pdb_info().pdb2pose("A", 100)</code>	Returns which residue in the pose corresponds to residue 100 of chain A in the pdb file
<code>pose.pdb_info().pose2pdb(25)</code>	Returns a string containing the residue and chain label in the pdb file corresponding to residue 25 of the pose
<code>CA_rmsd(pose1, pose2)</code>	Returns the root-mean-squared deviation of the location of C <sub>α</sub> atoms between the two poses

### Rosetta Scoring Terms

<code>fa_atr</code>	FA	van der Waals net attractive energy
<code>fa_rep</code>	FA	van der Waals net repulsive energy
<code>hbond_sr_bb, hbond_lr_bb</code>	FA/CEN	Hydrogen-bonding energies, short and long-range, backbone-backbone
<code>hbond_bb_sc, hbond_sc</code>	FA	Hydrogen-bonding energies, backbone-side-chain and side-chain-side-chain
<code>fa_sol</code>	FA	Solvation energies (Lazaridis-Karplus)
<code>fa_dun</code>	FA	Dunbrack rotamer probability
<code>fa_pair</code>	FA	Statistical residue-residue pair potential
<code>fa_intra_rep</code>	FA	Intraresidue repulsive Van der Waals energy
<code>fa_elec</code>	FA	Distance-dependent dielectric electrostatics
<code>pro_close</code>	FA	Proline ring closing energy
<code>dslf_ss_dst, dslf_cs_ang, dslf_ss_dih, dslf_ca_dih</code>	FA	Disulfide statistical energies (S-S distance, etc.)
<code>Ref</code>	FA/CEN	Amino acid reference energy of unfolded state
<code>p_aa_pp</code>	FA/CEN	Propensity of amino acid in (φ,ψ) bin, P(aa φ,ψ)
<code>Rama</code>	FA/CEN	Ramachandran propensities
<code>Vdw</code>	CEN	van der Waals "bumps" (repulsive only)
<code>Env</code>	CEN	Residue environment score (statistical)
<code>Pair</code>	CEN	Residue-residue pair score (statistical)
<code>Cbeta</code>	CEN	β-carbon score

<b>Rosetta: Scoring</b>	
<code>sf = ScoreFunction()</code>	Instantiates an empty <code>scorefxn</code> object from the <code>ScoreFunction</code> class
<code>sf = get_fa_scorefxn()</code>	Constructs a score function with the default Rosetta full-atom energy terms and weights
<code>sf = create_score_function("my_fxn")</code>	Constructs a score function with terms and weights from the <code>my_fxn</code> weights file
<code>sf = create_score_function_ws_patch("my_fxn", "docking")</code>	Constructs a score function from the <code>my_fxn</code> weights file with a patch for docking simulations
<code>sf.set_weight(fa_atr, 1.0)</code>	Sets the weight of the <code>fa_atr</code> term of the scoring function
<code>sf.get_weight(fa_atr)</code>	Gets the weight of the <code>fa_atr</code> term of the scoring function
<code>print sf</code>	Shows score function weights and details
<code>sf(pose)</code>	Returns the score of <code>pose</code> with the defined function <code>scorefxn</code> and stores the results in the <code>energies</code> object within <code>pose</code>
<code>sf.show(pose)</code>	Returns a table of weights and raw & weighted scores broken down by term
<code>pose.energies().show()</code>	Shows the breakdown of all energies (except backbone hydrogen-bonding energies) in the <code>pose</code> by residue
<code>pose.energies().show(5)</code>	Shows the breakdown of all energy contributions (except backbone hydrogen-bonding energies) from residue 5
<code>etable_atom_pair_energies(atom1, atom2, scorefxn)</code>	Returns a tuple of the attractive, repulsive, and solvation score components of a pair of <code>Atom</code> objects (not <code>AtomIDs</code> !)
<code>pose.energies().total_energies()[fa_atr]</code>	Returns the <code>fa_atr</code> contribution to the total energy
<code>pose.energies().residue_total_energies(5)[fa_atr]</code>	Returns the <code>fa_atr</code> contributions from residue 5
<code>hbond_set = hbonds.HBondSet()</code>	Instantiates an empty set for storing hydrogen-bonding energies and information
<code>pose.update_residue_neighbors()</code> <code>hbonds.fill_hbond_set(pose, False, hbond_set)</code>	Updates the <code>Energies</code> object within <code>pose</code> based on neighboring residues and fills <code>hbond_set</code> with this data (The option <code>False</code> is to forgo calculating a derivative.)
<code>hbond_set = get_hbonds(pose)</code>	Combines the steps above to instantiate and fill a set with hydrogen-bonding data
<code>hbond_set.show(pose)</code>	Shows a listing of all hydrogen bonds and their energies in a given <code>pose</code>
<code>hbond_set.show(pose, 5)</code>	Shows a listing of the hydrogen bonds and their energies from residue 5 of <code>pose</code>
<code>emap = EMapVector()</code>	Instantiates an energy map object to store a vector of scores
<code>scorefxn.eval_ci_2b(5, 6, pose, emap)</code>	Evaluates context-independent two-body energies between <code>pose</code> residues 5 and 6 and stores the energies in the energy map
<code>emap[fa_atr]</code>	Returns the <code>fa_atr</code> term from the map

<b>PyMOL Mover</b>	
<code>pmm = PyMOLMover()</code>	Instantiates the PyMOL mover
<code>pmm.apply(pose)</code>	Sends the pose coordinates to PyMOL for viewing
<code>pmm.send_energy(pose)</code>	Instructs PyMOL to color the pose by its total energy
<code>pmm.send_energy(pose, label=True)</code>	Instructs PyMOL to color the pose by its total energy and label each C $\alpha$ with the value.
<code>pmm.send_energy(pose, "fa_atr")</code>	Instructs PyMOL to color the pose by its fa_atr contribution
<code>pmm.label_energy(pose)</code>	Instructs PyMOL to label each C $\alpha$ with the value of its total energy contribution
<code>pmm.energy_type(fa_atr)</code> <code>pmm.update_energy(True)</code>	Sets the PyMOL mover to color by fa_atr every time the pose is updated with <code>pmm.apply(pose)</code>
<code>pmm.keep_history(True)</code>	Instructs PyMOL to store all pose conformations in separate frames
<code>colors = {1:"red", 2:"blue"}</code> <code>pmm.send_colors(pose, colors, "gray")</code>	Instructs PyMOL to color residue 1 red, 2 blue, and all others gray
<code>pmm.send_hbonds(pose)</code>	Instructs PyMOL to display distance lines for every hydrogen bond
<code>pmm.send_ss(pose)</code>	Uses DSSP to reassign secondary-structure and instructs PyMOL to display it as a cartoon
<code>pmm.send_polars(pose)</code>	Instructs PyMOL to color polar residues red and nonpolar residues blue
<code>pmm.send_movemap(pose, mm)</code>	Instructs PyMOL to color movable regions of the pose green and non-movable regions red
<code>pmm.send_foldtree(pose)</code>	Instructs PyMOL to color cutpoints red, jump points orange, and loop regions a unique color
<code>observer = AddPyMOLObserver(pose, TRUE)</code>	Updates PyMOL anytime a change is made to <code>pose</code> and keeps a history
<b>Residue Type Set Movers</b>	
<code>switch = SwitchResidueTypeSetMover("centroid")</code>	Instantiates a mover object that will change poses to the centroid residue type set ("fa_standard" is also available.)
<code>switch.apply(pose)</code>	Changes <code>pose</code> to the centroid residue type set
<code>recover_sidechains = ReturnSidechainMover(initial_fa_pose)</code>	Instantiates a mover object that will return the side chains and rotamers from an initial full-atom pose to a centroid version of the same peptide
<code>recover_sidechains.apply(pose)</code>	Changes <code>pose</code> to a full-atom type set and sets the rotamers to those found in <code>initial_fa_pose</code>

<b>MoveMap</b>	
<code>movemap = MoveMap()</code>	Instantiates a <code>movemap</code> object from the <code>MoveMap</code> class
<code>movemap.show(5)</code>	Displays the <code>movemap</code> settings for residues 1 to 5
<code>movemap.set_bb(True)</code>	Allows all backbone torsion angles to vary
<code>movemap.set_chi(True)</code>	Allows all side-chain torsion angles ( $\chi$ ) to vary
<code>movemap.set_bb(10, False)</code> <code>movemap.set_chi(10, False)</code>	Forbid the backbone and side-chain torsion angles of residue 10 from varying
<code>movemap.set_bb_true_range(10, 20)</code> <code>movemap.set_chi_true_range(10, 20)</code>	Allow backbone and side-chain torsion angles to vary in residues 10 to 20, inclusive
<code>movemap.set_jump(1, True)</code>	Allows jump #1 to be flexible

<b>Fragment Movers</b>	
<code>fragset = ConstantLengthFragSet(3)</code>	Constructs a 3-mer fragment set object
<code>fragset.read_fragment_file("fragfile")</code>	Loads data from the file <code>fragfile</code> into <code>fragset</code> (A fragment file must be downloaded from the Robetta server.)
<code>mover_3mer = ClassicFragmentMover(fragset, movemap)</code>	Constructs a fragment mover using the fragment set and the <code>movemap</code>
<code>mover_3mer.apply(pose)</code>	Replaces the angles in <code>pose</code> with those from a random 3-mer fragment from <code>fragset</code> , only in positions allowed by <code>movemap</code>
<code>smoothmover = SmoothFragmentMover(fragset, movemap)</code>	Constructs a "smooth" fragment mover (Fragment "insertions" are followed by a second, downstream fragment insertion chosen to minimize global disruption.)

<b>Small and Shear Movers</b>	
<code>kT = 1.0</code>	Variable simulating the product of the Boltzmann constant and temperature (1.0 approximates room temperature.)
<code>smallmover = SmallMover(movemap, kT, 5)</code> <code>shearmover = ShearMover(movemap, kT, 5)</code>	Construct a small or shear mover with a <code>movemap</code> , a temperature, and 5 moves
<code>smallmover.angle_max("H", 15)</code> <code>shearmover.angle_max("H", 15)</code>	Set the maximum change in dihedral angle within helix residues to 15° ("E" sets the maximum for sheet residues; "L" loop residues.)
<code>smallmover.apply(pose)</code> <code>shearmover.apply(pose)</code>	Apply the movers

<b>Minimize Mover</b>	
<code>minmover = MinMover()</code>	Constructs a minimize mover with default arguments
<code>minmover = MinMover(movemap, scorefxn, "linmin", 0.01, True)</code>	Construct a steepest descent minimize mover with a particular <code>MoveMap</code> and <code>ScoreFunction</code> and a score tolerance of 0.01
<code>minmover.movemap(movemap)</code>	Sets a movemap
<code>minmover.score_function(scorefxn)</code>	Sets a score function
<code>minmover.min_type("linmin")</code>	Sets a the minimization type to a line minimization (one direction in space), <i>i.e.</i> , "steepest descent"
<code>minmover.min_type("dfpmin")</code>	Sets a the minimization type to a David–Fletcher–Powell minimization (multiple iterations of "linmin" in conjugate directions)
<code>minmover.tolerance(0.5)</code>	Sets the mover to iterate until within 0.5 score points of the minimum
<code>minmover.apply(pose)</code>	Minimizes the pose
<b>Monte Carlo Object</b>	
<code>mc = MonteCarlo(pose, scorefxn, kT)</code>	Constructs a <code>MonteCarlo</code> object for a given pose and score function at a temperature of <code>kT</code>
<code>mc.set_temperature(1.0)</code>	Sets the temperature in the <code>MonteCarlo</code> object
<code>mc.boltzmann(pose)</code>	Accepts or rejects the current pose, compared to the last pose, according to the standard Metropolis criterion
<code>mc.show_scores()</code> <code>mc.show_counters()</code> <code>mc.show_state()</code>	Shows stored scores, counts of moves accepted/rejected, or both, respectively.
<code>mc.recover_low(pose)</code>	Sets the pose to the lowest-energy configuration ever encountered during the search
<code>mc.reset(new_pose)</code>	Resets all counters and sets the lowest and last pose stored to <code>new_pose</code> .
<b>Trial Mover</b>	
<code>smalltrial = TrialMover(smallmover, mc)</code>	Constructs a combination mover that will apply the small mover, then call the <code>MonteCarlo</code> object <code>mc</code> to accept or reject the new pose
<code>smalltrial.num_accepts()</code>	Returns the number of times the move was accepted
<code>smalltrial.acceptance_rate()</code>	Returns the acceptance rate of the moves

---

### Sequence Movers and Repeat Movers

<code>seqmover = SequenceMover()</code> <code>seqmover.addmover(smallmover)</code> <code>seqmover.addmover(shearmover)</code> <code>seqmover.addmover(minmover)</code>	Construct a combination mover that will call a series of other movers in sequence
<code>repeatmover = RepeatMover(fragmover, 10)</code>	Constructs a combination mover that will call <code>fragmover</code> 10 times
<code>randommover = RandomMover()</code> <code>randmover.addmover(smallmover)</code> <code>randmover.addmover(shearmover)</code> <code>randmover.addmover(minmover)</code>	Construct a combination mover that will randomly apply one of a set of movers each time it is applied

---

### Classic Relax Protocol

<code>relax = ClassicRelax()</code>	Instantiates an object that encompasses the entire standard Rosetta refinement protocol as presented in Bradley, Misura, & Baker 2005
<code>relax.set_scorefxn(scorefxn)</code>	Sets the score function
<code>relax.apply(pose)</code>	Applies the whole protocol

---

### Packer Task Object

<code>task = standard_packer_task(pose)</code>	Constructs a packer task object with instructions to repack all residues in <code>pose</code> using default rotamer library options, without repacking disulfide bonds
<code>task = TaskFactory.create_packer_task(pose)</code>	Constructs a default packer task object without any extra rotamer options
<code>task.restrict_to_repacking()</code>	Restricts all residues to repacking only (no design/"mutations")
<code>task.temporarily_fix_everything()</code>	Fixes/locks all residues' rotamers (no repacking)
<code>task.temporarily_set_pack_residue(5, True)</code>	Sets residue 5 to allow repacking
<code>generate_resfile_from_pdb("input.pdb", "output.resfile")</code> <code>generate_resfile_from_pose(pose, "output.resfile")</code>	Generate a resfile from a pdb file or a pose, respectively
<code>parse_resfile(pose, task, "file.resfile")</code>	Sets packer task for <code>pose</code> based on instructions in resfile

---

### Resfile Codes

NATRO	Use the native amino acid residue and native rotamer (do not repack)
NATAA	Use the <u>n</u> ative <u>a</u> mino <u>a</u> cid residue but allow repacking to other rotamers
PIKAA ILV	Pick from <u>a</u> mino <u>a</u> cid residues Ile, Leu, and Val and allow repacking
ALLAA	Use <u>a</u> ll <u>a</u> mino <u>a</u> cid residues and allow repacking

---

---

### Side Chain Packing Mover

<code>pack_mover = PackRotamersMover(scorefxn, task)</code>	Constructs a mover that will use instructions from the packer task to optimize or “mutate” side chain conformations in the pose
---	---

---

### Simple Point Mutation

<code>mutate_residue(pose, 49, 'E')</code>	Replaces residue 49 of pose with a glutamate (E) residue (does not optimize rotamers)
--	---

---

### Fold Tree

<code>ft = FoldTree()</code>	Constructs an empty fold tree
<code>ft = pose.fold_tree()</code>	Extracts the current fold tree from the <code>pose</code>
<code>pose.fold_tree(ft)</code>	Attaches the fold tree <code>ft</code> to the <code>pose</code>
<code>ft.add_edge(1, 30, -1)</code>	Creates a peptide edge (code -1) from residues 1 to 30 (This edge will build N-to-C)
<code>ft.add_edge(100, 31, -1)</code>	Creates a peptide edge from residues 100 to 31 (This edge will build C-to-N.)
<code>ft.add_edge(30, 100, 1)</code>	Creates a jump (rigid-body connection) between residues 30 and 100 (The jump number is 1; each jump needs a unique, sequential jump number.)
<code>ft.check_fold_tree()</code>	Returns <code>True</code> only for valid trees
<code>print ft</code>	Prints the fold tree
<code>ft.simple_tree(100)</code>	Creates a single-peptide-edge tree for a 100-residue protein
<code>ft.new_jump(40, 60, 50)</code>	Creates a jump from residue 40 to 60, a cutpoint between 50 and 51, and splits up the original edges to finish the tree
<code>ft.clear()</code>	Deletes all edges in the fold tree
<code>setup_foldtree(pose, "A_B", Vector1([1]))</code>	Creates a fold tree for <code>pose</code> with jump #1 between the centers of mass of chains A and B
<code>set_single_loop_fold_tree(pose, loop)</code>	Creates a fold tree for <code>pose</code> with jump points and a cutpoint defined by a <code>Loop</code> object, and splits up the original edges to finish the tree (See below for the <code>Loop</code> object.)

---

### Jump Object

<code>pose.jump(1).get_rotation()</code>	Returns the rotation matrix for the jump
<code>pose.jump(1).get_translation()</code>	Returns the translation vector for the jump

---

---

### Rigid Body Movers

<pre>pert_mover = RigidBodyPerturbMover(1, 8,     3)</pre>	Constructs a mover that will make a random rigid-body move of the downstream partner across jump #1 (Rotations and translations are chosen from a Gaussian with a mean of 8° and 3 Å, respectively.)
<pre>trans_mover = RigidBodyTransMover(pose,     jump_num) trans_mover.trans_axis(a) trans_mover.step_size(5)</pre>	Constructs a mover that will translate two partners, defined by jump_num, along an axis defined by vector a by 5 Å
<pre>spin_mover =     RigidBodySpinMover(jump_num) spin_mover.spin_axis(axis) spin_mover.rot_center(center) spin_mover.angle_size(45)</pre>	Constructs a mover that will spin the chain downstream of jump_num around a spin axis and rotation center by 45° (No specified angle size randomizes the spin.)
<pre>random_mover =     RigidBodyRandomizeMover(pose,     1, partner_upstream) random_mover =     RigidBodyRandomizeMover(pose,     1, partner_downstream)</pre>	Construct a mover that will rotate one of the partners across jump #1 randomly about its geometric center (partner_upstream and partner_downstream are predefined constants, not variables.)
<pre>slide = DockingSlideIntoContact(1) slide = FADockingSlideIntoContact(1)</pre>	Construct movers to translate two centroid or full-atom chains across jump #1 into contact, respectively

---

### Docking Protocols

<pre>dock_lowres = DockingLowRes(scorefxn_low,     jump_num)</pre>	Constructs a low-resolution, centroid-based Monte Carlo search protocol (50 rigid-body perturbations with adaptable step sizes)
<pre>dock_hires = DockMCMProtocol(scorefxn_high,     jump_num)</pre>	Constructs a high-resolution, full-atom-based Monte Carlo search protocol with rigid-body moves, side-chain packing, and minimization

---

### Loop Objects

<pre>loop = Loop(15, 24, 20)</pre>	Defines a loop with stems at residues 15 and 24, and a cutpoint at residue 20
<pre>loops = Loops()</pre>	Constructs an object to contain a set of loops
<pre>loops.add_loop(loop1)</pre>	Adds a Loop object to loops

---

**Loop Movers**

<code>ccd = CCDLoopClosureMover(loop1, movemap)</code>	Creates a mover which performs Canutescu & Dunbrack's cyclic coordinate descent loop closure algorithm
<code>loop_refine = LoopMover_Refine_CCD(loops)</code>	Creates a high-resolution refinement protocol consisting of cycles of small and shear moves, side-chain packing, CCD loop closure, and minimization.

---



---

**RMSD-Calculating Functions**

<code>CA_rmsd(pose1, pose2)</code>	Returns the RMSD between the C $\alpha$ atoms of pose1 and pose2
<code>calc_Lrmsd(pose1, pose2, Vector([1]))</code>	Return the ligand RMSD between pose1 and pose2
<code>loop_rmsd(pose, ref_pose, loops, True)</code>	Returns the RMSD of all loops in the reference frame of the fixed protein structure

---



---

**Job Distributor**

<code>jd = PyJobDistributor("output", 10, scorefxn)</code>	Constructs a job distributor that will create 10 model structures named output_1.pdb to output_9.pdb and a file containing a table of scores
<code>jd.native_pose = native_pose</code>	Sets the native pose for RMSD comparisons
<code>jd.job_complete</code>	Returns True if all decoys have been output
<code>jd.output_decoy(pose)</code>	Outputs pose to a file and increments the decoy number
<code>while not jd.job_complete: # Code for creating decoys jd.output_decoy(pose)</code>	Loop to create decoys until all have been output
<code>jd.additional_decoy_info = "Created by Andy"</code>	Sets a string to be output to the pdb files

---