

Appendix A: Command Reference

Python Commands and Syntax	
<pre>i = 1 j = "Bob"</pre>	Variable assignments
<pre>print j, " thinks ", i, " = 0."</pre>	Prints Bob thinks 1 = 0.
<pre>for i in range(1,10): print i</pre>	The newly defined variable <code>i</code> ranges from 1 up to,(but not including) 10 and the command <code>print i</code> is executed for each value.
<pre>if x < 0: x = 0 print elif x==0: print "zero" else: print "positive"</pre>	Conditional statement that executes lines only if Boolean statements are true. Use indenting to indicate blocks of code executed together under the conditional
<pre>def myfunc(a, b) # code here return c,d,e</pre>	Defines a function. Also acceptable, <code>return(c, d, e)</code> , but not <code>return[c, d, e]</code>
<pre>returned_values = myfunc(a, b) value_of_c = returned_values[0] value_of_d = returned_values[1] value_of_e = returned_values[2]</pre>	Syntax for using multiple values returned by a function called with variables a and b.
<pre>outfile = open('out.txt','w') print >>outfile "hello" outfile.close()</pre>	Prints hello to a new file named <code>out.txt</code>
<pre>outfile.write(str(i)+";"+str(score) +"\\n")</pre>	Alternate way to write to a (previously opened) file
Python Math	
<pre>import math</pre>	Imports math functions from Python
<pre>math.exp(5)</pre>	Returns the value of e^5
<pre>import random</pre>	Imports random number functions from Python
<pre>random.random()</pre>	Returns a random float between 0 and 1
<pre>random.randint(5,10)</pre>	Returns a random integer between 5 and 10 (inclusive)
<pre>random.gauss(5,10)</pre>	Returns a random number from a Gaussian distribution with a median of 5 and a standard deviation of 10
Rosetta: Vector	
<pre>v = numeric.xyzVector(x,y,z)</pre>	Creates an xyz vector used for Cartesian coordinates
<pre>print v</pre>	Prints v and its elements
<pre>print v.x, v.y, v.z</pre>	
<pre>v.norm</pre>	L2 vector norm of v
<pre>v.dot(v2)</pre>	Dot product of v and v2
<pre>v.cross(v2)</pre>	Cross product of v and v2

Rosetta: Pose Object

<code>pose = Pose()</code>	Creates a an empty pose object.
<code>pose_from_pdb(pose, "/path/to/ input_file.pdb")</code>	Loads the pdb file into the pose object.
<code>pose = Pose("lyfp.pdb")</code>	Creates new object directly from the pdb file.
<code>make_pose_from_sequence(pose, "AAAAAA", "fa_standard")</code>	Loads a pose from the given sequence string using standard all-atom residue type templates
<code>print pose</code>	Displays PDB filename, sequence, and fold tree
<code>pose.assign(otherpose)</code>	Copies <code>otherpose</code> onto <code>pose</code> . You cannot simply write <code>pose = otherpose</code> , as that will only point <code>pose</code> to <code>otherpose</code> and not actually copy it.
<code>dump_pdb(pose, "/path/to/output_file.pdb")</code>	Creates pdb file named <code>output_file.pdb</code> using information from pose object.
<code>pose.total_residue()</code>	Returns number of residues in pose
<code>pose.phi(5)</code>	Returns the ϕ or ψ angle of the 5 th residue in the pose; returns 2 nd χ of the 5 th residue
<code>pose.psi(5)</code>	
<code>pose.chi(2,5)</code>	
<code>pose.set_phi(5,60.0)</code>	Sets the ϕ or ψ angle of the 5 th residue in pose to 60°; sets the 2 nd χ of the 5 th residue to 60°
<code>pose.set_psi(5,60.0)</code>	
<code>pose.set_chi(2,5,60.0)</code>	
<code>print pose.residue(5)</code>	Prints the amino acid details of residue 5
<code>print pose.residue(5).xyz("CA")</code>	Prints the <code>numeric.xyzVector</code> for the second atom (CA) of residue 5
<code>print pose.residue(5).xyz(2)</code>	
<code>pose.conformation().set_bond_length(atom1, atom2,length)</code>	Sets the bond length between objects <code>atom1</code> and <code>atom2</code> to a value of <code>length</code> .
<code>pose.conformation().set_bond_angle(atom1, atom2,atom3,bond_angle)</code>	Sets the bond angle of objects <code>atom1</code> , <code>atom2</code> and <code>atom3</code> to a value of <code>bond_angle</code> .
<code>atomN = pose.residue(5).atom('N')</code>	Creates a pointer to the N atom object of residue 5
<code>coord = atomN.xyz()</code>	Prints the Cartesian coordinates of <code>atomN</code>
<code>print coord</code>	
<code>print coord.x, coord.y, coord.z</code>	
<code>NCbond = atomN.xyz() - atomC.xyz()</code>	Calculates and prints the distance between <code>atomN</code> and <code>atomC</code>
<code>print NCbond.norm()</code>	
<code>for i in range (1,pose.total_residue()+1): <command> # on pose.residue(i)</code>	Loops through all residues in pose and runs <code><command></code> on each one
<code>pose.pdb_info().name()</code>	Gives the name of the PDB file input to pose
<code>pose.pdb_info().number(i)</code>	Gives the PDB number of residue <code>i</code>
<code>pose.pdb_info().chain(i)</code>	Gives the PDB chain of residue <code>i</code>
<code>pose.pdb_info().icode(i)</code>	Gives the PDB insert code of residue <code>i</code>
<code>pose.pdb_info().pdb2pose("A",100)</code>	Gives the pose's internal residue index
<code>pose.pdb_info().pose2pdb(25)</code>	Gives the PDB chain/number/insert code from pose's internal residue index
<code>print CA_rmsd(pose1, pose2)</code>	Calculates and prints the root-mean-squared deviation of the location of C _α atoms between the two poses

Rosetta: Scoring

<code>scorefxn = create_score_function('standard')</code>	Defines a score function with standard full-atom energy terms and weights
<code>scorefxn2=core.scoring.ScoreFunction() scorefxn2.set_weight(core.scoring.fa_atr, 1.0)</code>	Copies the score function and alters the weight of the <code>fa_atr</code> term.
<code>print scorefxn</code>	Shows score function weights and details
<code>scorefxn(pose)</code>	Returns the score of pose with the defined function <code>scorefxn</code> .
<code>scorefxn.show(pose)</code>	Returns the weights, raw scores, and weighted scores of the pose broken down by scoring term
<code>pose.energies().show() pose.energies().show(resnum)</code>	Shows the breakdown of the energies by residue
<code>emap = rosetta.core.scoring.TwoBodyEMapVector ()</code>	Creates an energy map object to store a vector of scores
<code>scorefxn.eval_ci_2b(rsd1,rsd2,pose,emap)</code>	Evaluates context-independent two-body energies between residues <code>rsd1</code> and <code>rsd2</code> and stores the energies in the energy map
<code>print emap[rosetta.core.scoring.fa_atr]</code>	Print <code>fa_atr</code> term from the energy map
<code>hbond_set = rosetta.core.scoring.hbonds.HBondSet()</code>	Creates an HBond set object for storing hydrogen bonding information
<code>pose.update_residue_neighbors(); rosetta.core.scoring.hbonds. fill_hbond_set(pose,False,hbond_set)</code>	Stores H-bond info from pose in the <code>Hbond_set</code> object.
<code>hbond_set.show(pose)</code>	Prints H-bond info from the <code>hbond_set</code>
<code>calc_total_sasa(pose, 1.5)</code>	Calculates the total solvent-accessible surface area using a 1.5 Å probe

Rosetta Full-atom Scoring Functions

<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 bonds, short and long-range (backbone-backbone)
<code>hbond_bb_sc, hbond_sc</code>	FA	Hydrogen bonds (backbone-side chain and side chain-side chain)
<code>fa_sol</code>	FA	Solvation (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
<code>hack_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, $P(aa \phi,\psi)$
<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

Residue Type Set Mover

<code>switch =</code> <code> SwitchResidueTypeSetMover('centroid')</code>	Creates a mover which will change poses to the centroid residue type set (<code>fa_standard</code> also available)
<code>switch.apply(pose)</code>	Changes pose to the centroid residue types

MoveMap

<code>movemap = MoveMap()</code>	Creates a MoveMap
<code>movemap.show(Nres)</code>	Prints the MoveMap contents for residues 1 to Nres
<code>movemap.set_bb(True)</code>	Allows all backbone torsion angles to vary when movemap is applied
<code>movemap.set_chi(True)</code>	Allows all side chain torsion angles (χ) to vary when movemap is applied
<code>movemap.set_bb(10,False)</code> <code>movemap.set_chi(10,False)</code>	Forbid residue 10's backbone and side chain torsion angles from varying
<code>movemap.set_bb_true_range(10,20)</code>	Allows backbone torsion angles to vary in residues 10 to 20, inclusive; sets all other residues to False.
<code>movemap.set_jump(1, True)</code>	Allows jump #1 to be flexible

Fragment Movers

<code>fragset = ConstantLengthFragSet(3,</code> <code> "aatestA03_05.200_v1_3")</code>	Creates a fragment set and loads the fragments from the data file
<code>mover_3mer = ClassicFragmentMover(fragset,movemap)</code>	Creates a fragment mover using the fragset and the movemap
<code>mover_3mer.apply(pose)</code>	Inserts a random 3-mer fragment from the fragset into the pose, only in positions allowed by the movemap
<code>smoothmover =</code> <code> SmoothFragmentMover(fragset,</code> <code> movemap)</code>	Fragment insertions are followed by a second, downstream fragment insertion chosen to minimize global disruption

Small and Shear Movers

<code>kT = 1.0</code> <code>n_moves = 1</code> <code>smallmover = SmallMover(movemap,kT,n_moves)</code> <code>shearmover = ShearMover(movemap,kT,n_moves)</code>	Creates a small or shear mover with a movemap, a temperature, and the number of moves
<code>smallmover = SmallMover()</code> <code>shearmover = ShearMover()</code>	Default settings are all backbone moves allowed, $kT = 0.5$, and $n_moves = 1$
<code>smallmover.apply(pose)</code> <code>shearmover.apply(pose)</code>	Applies the movers

Minimize Mover

<code>minmover = MinMover()</code>	Creates a minimize mover with default arguments
<code>minmover = MinMover(movemap, scorefxn, min_type, tolerance, True)</code>	Creates a minimize mover with a particular MoveMap, ScoreFunction, minimization type, or score tolerance
<code>minmover.movemap(movemap)</code>	Set a movemap
<code>minmover.score_function(scorefxn)</code>	Set a scorefunction
<code>minmover.min_type('linmin')</code>	Set a the minimization type to a line minimization (one direction in the space)
<code>minmover.min_type('dfpmin')</code>	Set a the minimization type to a David-Fletcher-Powell minimization (multiple iterations of linmin in conjugate directions)
<code>minmover.tolerance(0.5)</code>	Set the mover to iterate until within 0.5 score points of the minimum
<code>minmover.apply(pose)</code>	Apply the minmover to a pose

MonteCarlo

<code>mc = MonteCarlo(pose, scorefxn, kT)</code>	Creates a MonteCarlo object
<code>mc.set_temperature(1.0)</code>	Sets the temperature in the MonteCarlo object
<code>mc.boltzmann(pose)</code>	Accepts or rejects the pose object, compared to the pose last time the mc object was called, according to the standard Metropolis criterion.
<code>mc.show_scores()</code>	Shows stored scores, counts of moves
<code>mc.show_counters()</code>	accepted/rejected, or both.
<code>mc.show_state()</code>	
<code>mc.recover_low(pose)</code>	Sets the pose to the lowest-energy configuration ever seen during the search
<code>mc.reset(pose)</code>	Resets all counters and sets the low- and last-pose to the current pose state.

TrialMover

<code>smalltrial = TrialMover(smallmover, mc)</code>	Creates a mover which will apply the small mover, then call the MonteCarlo object mc. This mover will also give more explicit tags for the <code>mc.show_state()</code> output.
<code>smalltrial.num_accepts()</code>	Number of times the move was accepted
<code>smalltrial.acceptance_rate()</code>	Acceptance rate of the moves

SequenceMover and RepeatMover

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

Side Chain Packing Movers

<code>pack_mover = PackRotamersMover(scorefxn, task) pack_mover.apply(pose)</code>	Creates a mover that will use instructions from the <code>task</code> to do packing to optimize side chain conformations in the pose
<code>rot_trial = RotamerTrials(scorefxn, task) rot_trial.apply(pose)</code>	Creates a mover that will use instructions from the <code>task</code> to do Rotamer Trials to optimize side chain conformations in the pose
<code>task = standard_packer_task(pose)</code>	Configures a packer task to pack all residue using default rotamer library options for extra χ angles from the command-line initialization, and not repacking disulfide bonds
<code>task = TaskFactory.create_packer_task(pose)</code>	Creates a vanilla packer task based on a pose, without any extra rotamer options
<code>task.or_include_current(True)</code>	Includes current rotamers in pose to packer
<code>task.restrict_to_repacking()</code>	Restricts all residues to repacking (no design)
<code>task.fix_everything()</code>	Sets all residues to no repacking
<code>task.set_pack_residue(i)</code>	Sets residue <code>i</code> to allow repacking
<code>task.read_resfile("resfile")</code>	Sets task based on instructions in resfile
<code>generate_resfile_from_pdb(test.pdb, "resfile")</code>	Generates a resfile from a pdb file or a pose, respectively
<code>generate_resfile_from_pose(pose, "resfile")</code>	

Fold Tree

<code>ft = FoldTree() ft = pose.fold_tree()</code>	Extracts the current fold tree from the pose
<code>pose.fold_tree(ft)</code>	Attaches the fold tree <code>ft</code> into the pose.
<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.
<code>ft.add_edge(100,101,2)</code>	Creates a second jump between residues 100 and 101. The jump number is 2. 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 residues 40 to 60, a cutpoint between 50 and 51, and splits up the original edges as needed to finish the tree.
<code>ft.clear()</code>	Deletes all edges in the fold tree.

Rigid Body Movers

<pre>pert_mover = RigidBodyPerturbMover(jump_num,3,8) pert_mover.apply(pose)</pre>	Makes a random rigid body move of the downstream partner. Random rotation chosen from a Gaussian of standard deviation of 8°, and translation chosen from a Gaussian of standard deviation 3 Å
<pre>transmover = RigidBodyTransMover(pose, jump_num) transmover.trans_axis(a) transmover.step_size(5) transmover.apply(pose)</pre>	Creates a mover that will translate two partners, defined by <code>jump_num</code> , along an axis defined by <code>numeric.xyzVector a</code> , by 5 Å.
<pre>spinmover = RigidBodySpinMover(jump_num) spinmover.spin_axis(axis) spinmover.rot_center(center) spinmover.angle_size(45)</pre>	Creates a mover that will spin the residues downstream of <code>jump_num</code> around a spin axis and rotation center (of type <code>numeric.xyzVector</code>) by 45°. No specified <code>angle_size</code> randomizes the spin.

Docking Movers

<pre>DockingProtocol() DockingProtocol().setup_foldtree(pose) DockingProtocol().setup_foldtree(pose,'HL_A')</pre>	Protocol for a full, multiscale docking run Sets up a fold tree for docking chains H and L relative to chain A
<pre>movemap = MoveMap() movempa.set_jump(jump_num,True) minmover = MinMover() minmover.movemap(movemap)</pre>	Sets up a mover to minimize over the rigid-body coordinates
<pre>dock_lowres = DockingLowRes(scorefxn_low, jump_num) dock_lowres.apply(pose)</pre>	Low-resolution, centroid based MC search (50 RigidBodyPerturbMoves with adaptable step sizes)
<pre>dock_hires = DockingHighRes(scorefxn_high, jump_num) dock_hires.apply(pose)</pre>	High-resolution, all-atom based MCM search with rigid-body moves, side-chain packing, and minimization
<pre>cs = ConformerSwitchMover(start,end, jump_num,scorefxn,"laaa.pdb") cs.apply(pose)</pre>	Picks a new backbone conformation from the ensemble (conformer selection docking). <code>start</code> and <code>end</code> indicate residue number range for backbone swapping.
<pre>randomize1 = RigidBodyRandomizeMover(pose, jump_num, partner_upstream) randomize2 = RigidBodyRandomizeMover(pose, jump_num, partner_downstream)</pre>	When applied, globally randomizes the rotation of the upstream partner. When applied, globally randomizes the rotation of the downstream partner.
<pre>DockingProtocol().calc_Lrmsd(pose1, pose2)</pre>	Calculates RMSD of smaller partner after superposition of larger partner

Loops

<code>loop1 = Loop(15,24,20)</code>	Defines a loop with stems at residues 15 and 24, and a cut point at residue 20
<code>loops = Loops() loops.add_loop(loop1)</code>	Creates an object to contain a set of loops
<code>set_single_loop_fold_tree(pose, loop)</code>	Sets the pose's fold tree for single-loop optimization
<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.
<code>Lrms = loop_rmsd(pose,reference_pose, loops, True)</code>	Calculates the rmsd of all loops in the reference frame of the fixed protein structure

Job Distributor

<code>jd = JobDistributor("output", 1000, scorefxn_high)</code>	Creates a job distributor which will create 1000 model structures named <code>output_1.pdb</code> to <code>output1000.pdb</code> . Files include <code>scorefxn_high</code> energies.
<code>Pose native_pose("1aaa.pdb") jd.native_pose = native_pose</code>	Sets the native pose (loaded from <code>1aaa.pdb</code>) for rmsd comparisons
<code>jd.job_complete</code>	Boolean indicating whether all decoys have been output.
<code>jd.output_decoy(pose)</code>	Outputs the pose to a file and increments the decoy number.
<code>while (jd.job_complete == False): #[create the decoy called pose] jd.output_decoy(pose)</code>	Loop to create decoys until all have been output
<code>jd.additional_info = "Created by Andy"</code>	Sets a string to be output to the pdb file
