Appendix A: Command Reference

Python Commands and Syntax	
`# I am a comment; Python ignores me.`	Comments

`"""I am also ignored.`
`Me too!`
`"""`
	Block comments (doc strings)

`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`
	Simple variable assignments (Python is case sensitive.)

`tuple = (1, 2, 3)`
`list = [15, 'X', -1.5, "lion"]`
`2_D_list[[1, 0], [0, 1]]`
`dict = {"apple": "sour", "days": 24}`
	Assignment to various sequences (tuples, lists, and dictionaries)

`print i + 2, 3/2, float(3)/2,` `3.0/2.0, 2*j, (i + 3)**2`	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.)
`print k + " thinks " + str(i) +` `" = 0."`	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.)

`print list[1], tuple[1:2], k[0:2],`
`dict["days"],`
`2_d_list[0][1]`
	Outputs `15 (2, 3) Bo 24 0`. (Lists, strings, *etc.*, are indexed starting with 0.)

`i += 1`
`j -= 1`
`print i, j`
	Increment and decrement operators.
	Outputs `2 0.0`

`tuple[2] = 4` `first_name[0] = 'R'`	Raise errors; tuples and strings are immutable.
`list[3] = "tiger"` `dict["apple"] = "sweet"`	Lists and dictionaries are mutable.

`list.append(False)`
`dict["new entry"] = 'Z'`
`print list, dict`
	Outputs `[15, 'X', -1.5, "tiger", False]`
	`{"apple": "sweet", "days": 24, "new entry", 'Z'}`

`for i in range(1, 10):` `print i`	The newly defined variable `i` ranges from 1 up to *but not including* 10, and the command `print i` is executed for each value.
`for j in ("cats", "dogs", "fish"):` `print j`	Outputs: **`cats`** **`dogs`** **`fish`**

`if x < 0:`
`print "negative"`
`elif x == 0:`
`print "zero"`
`else:`
`print "positive"`
	Conditional statement that executes lines only if Boolean statements are true.
	`elif` means "or else, check if".
	Do not mix up = and ==!
	Use indenting to indicate blocks of code executed together under the conditional.

`if i_am_a_boolean:`
`  print "Yes!"`
`if not i_am_an_integer:`
`  print "No!"`
	Outputs:
	`Yes!`
	`No`!

`a, b = 1, 1`
`print a is b`
`print a == b`
	Outputs:
	`False`
	`True`
	(a and b contain the same value, but are not two names for the same object!)

`while len(list) <= 7:`
`  list.append("blah")`
`print list`
	Outputs:
	`[15, 'X', -1.5, "tiger", False, "blah", "blah", "blah"]`

`def my_func(a, b):`
`  c = a + b`
`  d = b + c`
`  e = c + d`
`  return c, d, e`
	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)`

`returned_values = my_func(1, 2)`
`value_of_c = returned_values[0]`
`value_of_d = returned_values[1]`
`value_of_e = returned_values[2]`
	Syntax for using multiple values returned by a function, *e.g.*, `value_of_c` is `3`.

`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`
	Defines a class with two methods

`circle = MyCircle()`
`circle.draw()`
`circle.radius = 1.5`
`circle.draw(1)`
	Constructs a `MyCircle` object, draws a circle in color 0, and then draws a circle in color 1

`file = open("out.txt", 'w')`
`file.write("hello")`
`file.close()`
	Opens a new file named `out.txt` for writing and outputs `hello` to the file. (Be sure to close your files when finished with them.)

`import module`	Imports and runs the `module` module.py so that its functions can be called with `module.function()`.
`from module import function`	Imports the specific function `function` so that it can be called with simply `function()`.
`from module import *`	Imports all of the (public) functions from `module.py`.

Python Math	
`math.exp(5)`	Returns the value of e^5
`math.pi`	Returns the value of π
`math.sin(theta)`	Returns the value of sin θ, where θ is in radians
`math.acos(x)`	Returns the value of arcos x in radians
`math.degrees(rad)`	Converts radians to degrees
`meth.radians(deg)`	Converts degrees to radians
`random.random()`	Returns a random floating point number between 0.0 and 1.0
`random.randint(5, 10)`	Returns a random integer between 5 and 10 (inclusive)
`random.gauss(10, 2)`	Returns a random number from a Gaussian distribution with a mean of 10 and a standard deviation of 2

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

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

Rosetta: Pose Object	
`pose = Pose()`	Instantiates an empty `pose` object from the `Pose` class
`pose = pose_from_pdb("input_file.pdb")`	Loads a pdb file from the working directory into a new `pose` object
`pose = pose_from_rcsb("1YY8")`	Loads pdb 1YY8 from the Internet
`pose = pose_from_sequence("AAAAAA", "fa_standard")`	Creates a new pose from the given sequence using standard, full-atom residue type templates
`print pose`	Displays information about the pose object: pdb filename, sequence, and fold tree
`pose.sequence()`	Returns the sequence of the pose structure
`get_secstruct(pose)`	Assigns secondary structure information to `pose` and outputs it to the screen
`pose.assign(other_pose)`	Copies `other_pose` onto `pose`. You cannot simply type `pose = other_pose`, as that will only point `pose` to `other_pose` and not actually copy it.
`pose.dump_pdb("output_file.pdb")`	Creates a pdb file named `output_file.pdb` in the working directory using information from pose object.
`pose.is_fullatom()`	Returns `True` if the pose contains a full-atom representation of a structure
`pose.total_residue()`	Returns total number of residues in the pose

`pose.phi(5)`
`pose.psi(5)`
`pose.omega(5)`
`pose.chi(2, 5)`
	Returns the φ, ψ, ω, or χ_2 angles (in degrees) of the 5th residue in the pose

`pose.set_phi(5, 60.0)`
`pose.set_psi(5, 60.0)`
`pose.set_omega(5, 60.0)`
`pose.set_chi(2, 5, 60.0)`
	Sets the φ, ψ, ω, or χ_2 angles of the 5th residue in the pose to 60.0°

`print pose.residue(5)`	Outputs the amino acid details of residue 5
`pose.residue(5).name()`	Returns the 3-letter residue name for residue 5

`pose.residue(5).is_polar()`
`pose.residue(5).is_aromatic()`
`pose.residue(5).is_charged()`
	Return `True` if the 5th residue is of the queried type

`pose.residue(5).xyz("CA")` `pose.residue(5).xyz(2)`	Return the displacement vector of the α carbon (CA) of residue 5, which is the 2nd atom listed for that residue in the pose and a standard pdb file
`pose.residue(5).atom_index("CA")`	Returns `2`
`for i in range (1, pose.total_residue() + 1):` `  print pose.residue(i).name()`	Loops through all residues in the pose and outputs the 3-letter name of each (Unlike Python, Rosetta indexes residues starting with 1.)
`atom = pose.residue(5).atom("CA")`	Constructs an atom object for the α carbon (CA) of residue 5

`atom1 = AtomID(1, 5)`
`atom2 = AtomID(2, 5)`
`atom3 = AtomID(3, 5)`
	Construct unique atom *identifier* objects for the 1st, 2nd, and 3rd, atoms of *any* residue 5, respectively (This is not the same as the above command!)

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

Rosetta Scoring Terms		
`fa_atr`	FA	van der Waals net attractive energy
`fa_rep`	FA	van der Waals net repulsive energy
`hbond_sr_bb, hbond_lr_bb`	FA/CEN	Hydrogen-bonding energies, short and long-range, backbone–backbone
`hbond_bb_sc, hbond_sc`	FA	Hydrogen-bonding energies, backbone–side-chain and side-chain–side-chain
`fa_sol`	FA	Solvation energies (Lazaridis–Karplus)
`fa_dun`	FA	Dunbrack rotamer probability
`fa_pair`	FA	Statistical residue–residue pair potential
`fa_intra_rep`	FA	Intraresidue repulsive Van der Waals energy
`fa_elec`	FA	Distance-dependent dielectric electrostatics
`pro_close`	FA	Proline ring closing energy
`dslf_ss_dst, dslf_cs_ang, dslf_ss_dih, dslf_ca_dih`	FA	Disulfide statistical energies (S–S distance, *etc.*)
`ref`	FA/CEN	Amino acid reference energy of unfolded state
`p_aa_pp`	FA/CEN	Propensity of amino acid in (φ,ψ) bin, P(aa|φ,ψ)
`rama`	FA/CEN	Ramachandran propensities
`vdw`	CEN	van der Waals “bumps” (repulsive only)
`env`	CEN	Residue environment score (statistical)
`pair`	CEN	Residue–residue pair score (statistical)
`cbeta`	CEN	β-carbon score

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

`scorefxn.eval_ci_2b(5, 6, pose, emap)`	Evaluates context-independent two-body energies between pose residues 5 and 6 and stores the energies in the energy map
`emap[fa_atr]`	Returns the `fa_atr` term from the map

PyMOL Mover	
`pmm = PyMOL_Mover()`	Instantiates the PyMOL mover
`pmm.apply(pose)`	Sends the pose coordinates to PyMOL for viewing
`pmm.send_energy(pose)`	Instructs PyMOL to color the pose by its total energy
`pmm.send_energy(pose, label=True)`	Instructs PyMOL to color the pose by its total energy and label each Cα with the value.
`pmm.send_energy(pose, "fa_atr")`	Instructs PyMOL to color the pose by its fa_atr contribution
`pmm.label_energy(pose)`	Instructs PyMOL to label each Cα with the value of its total energy contribution
`pmm.energy_type(fa_atr)` `pmm.update_energy(True)`	Sets the PyMOL mover to color by fa_atr every time the pose is updated with `pmm.apply(pose)`
`pmm.keep_history(True)`	Instructs PyMOL to store all pose conformations in separate frames
`colors = {1:"red", 2:"blue"}` `pmm.send_colors(pose, colors, "gray")`	Instructs PyMOL to color residue 1 red, 2 blue, and all others gray
`pmm.send_hbonds(pose)`	Instructs PyMOL to display distance lines for every hydrogen bond
`pmm.send_ss(pose)`	Uses DSSP to reassign secondary-structure and instructs PyMOL to display it as a cartoon
`pmm.send_polars(pose)`	Instructs PyMOL to color polar residues red and nonpolar residues blue
`pmm.send_movemap(pose, mm)`	Instructs PyMOL to color movable regions of the pose green and non-movable regions red
`pmm.send_foldtree(pose)`	Instructs PyMOL to color cutpoints red, jump points orange, and loop regions a unique color
`pmm.view_foldtree_diagram(pose)`	Draws a 3-D fold tree diagram in PyMOL
`observer = PyMOL_Observer(pose, True)`	Updates PyMOL anytime a change is made to `pose` and keeps a history

Residue Type Set Movers	
`switch = SwitchResidueTypeSetMover("centroid")`	Instantiates a mover object that will change poses to the centroid residue type set (`"fa_standard"` is also available.)
`switch.apply(pose)`	Changes `pose` to the centroid residue type set
`recover_sidechains = ReturnSidechainMover(initial_fa_pose)`	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
`recover_sidechains.apply(pose)`	Changes `pose` to a full-atom type set and sets the rotamers to those found in `initial_fa_pose`

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

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

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

Minimize Mover	
`minmover = MinMover()`	Consructs a minimize mover with default arguments
`minmover = MinMover(movemap, scorefxn, "linmin", 0.01, True)`	Construct a steepest descent minimize mover with a particular `MoveMap` and `ScoreFunction` and a score tolerance of 0.01
`minmover.movemap(movemap)`	Sets a movemap
`minmover.score_function(scorefxn)`	Sets a score function
`minmover.min_type("linmin")`	Sets a the minimization type to a line minimization (one direction in space), *i.e.*, "steepest descent"
`minmover.min_type("dfpmin")`	Sets a the minimization type to a David–Fletcher–Powell minimization (multiple iterations of `"linmin"` in conjugate directions)
`minmover.tolerance(0.5)`	Sets the mover to iterate until within 0.5 score points of the minimum
`minmover.apply(pose)`	Minimizes the pose

Monte Carlo Object	
`mc = MonteCarlo(pose, scorefxn, kT)`	Constructs a `MonteCarlo` object for a given pose and score function at a temperature of `kT`
`mc.set_temperature(1.0)`	Sets the temperature in the `MonteCarlo` object
`mc.boltzmann(pose)`	Accepts or rejects the current pose, compared to the last pose, according to the standard Metropolis criterion

`mc.show_scores()`
`mc.show_counters()`
`mc.show_state()`
	Shows stored scores, counts of moves accepted/rejected, or both, respectively.

`mc.recover_low(pose)`	Sets the pose to the lowest-energy configuration ever encountered during the search
`mc.reset(new_pose)`	Resets all counters and sets the lowest and last pose stored to `new_pose`.

Trial Mover	
`smalltrial = TrialMover(smallmover, mc)`	Constructs a combination mover that will apply the small mover, then call the `MonteCarlo` object `mc` to accept or reject the new pose
`smalltrial.num_accepts()`	Returns the number of times the move was accepted
`smalltrial.acceptance_rate()`	Returns the acceptance rate of the moves

Sequence Movers and Repeat Movers	

`seqmover = SequenceMover()`
`seqmover.addmover(smallmover)`
`seqmover.addmover(shearmover)`
`seqmover.addmover(minmover)`
	Construct a combination mover that will call a series of other movers in sequence

`repeatmover = RepeatMover(fragmover, 10)`	Constructs a combination mover that will call `fragmover` 10 times

`randommover = RandomMover()`
`randmover.addmover(smallmover)`
`randmover.addmover(shearmover)`
`randmover.addmover(minmover)`
	Construct a combination mover that will randomly apply one of a set of movers each time it is applied

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

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

Resfile Codes	
`NATRO`	Use the native amino acid residue and native rotamer (do not repack)
`NATAA`	Use the native amino acid residue but allow repacking to other rotamers
`PIKAA ILV`	Pick from amino acid residues Ile, Leu, and Val and allow repacking
`ALLAA`	Use all amino acid residues and allow repacking

Side Chain Packing Mover	
`pack_mover = PackRotamersMover(scorefxn, task)`	Constructs a mover that will use instructions from the packer task to optimize or "mutate" side chain conformations in the pose

Simple Point Mutation	
`mutate_residue(pose, 49, 'E')`	Replaces residue 49 of pose with a glutamate (E) residue (does not optimize rotamers)

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

Jump Object	
`pose.jump(1).get_rotation()`	Returns the rotation matrix for the jump
`pose.jump(1).get_translation()`	Returns the translation vector for the jump

Rigid Body Movers	
`pert_mover = RigidBodyPerturbMover(1, 8, 3)`	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.)

`trans_mover = RigidBodyTransMover(pose, jump_num)`
`trans_mover.trans_axis(a)`
`trans_mover.step_size(5)`
	Constructs a mover that will translate two partners, defined by jump_num, along an axis defined by vector a by 5 Å

`spin_mover = RigidBodySpinMover(jump_num)`
`spin_mover.spin_axis(axis)`
`spin_mover.rot_center(center)`
`spin_mover.angle_size(45)`
	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.)

`random_mover = RigidBodyRandomizeMover(pose, 1, partner_upstream)` `random_mover = RigidBodyRandomizeMover(pose, 1, partner_downstream)`	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.)
`slide = DockingSlideIntoContact(1)` `slide = FADockingSlideIntoContact(1)`	Construct movers to translate two centroid or full-atom chains across jump #1 into contact, respectively

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

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

Loop Movers	
`ccd = CCDLoopClosureMover(loop1, movemap)`	Creates a mover which performs Canutescu & Dunbrack's cyclic coordinate descent loop closure algorithm
`loop_refine = LoopMover_Refine_CCD(loops)`	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	
`CA_rmsd(pose1, pose2)`	Returns the RMSD between the Cα atoms of `pose1` and `pose2`
`calc_Lrmsd(pose1, pose2, Vector([1]))`	Return the ligand RMSD between `pose1` and `pose2`
`loop_rmsd(pose, ref_pose, loops, True)`	Returns the RMSD of all loops in the reference frame of the fixed protein structure

Job Distributor	
`jd = PyJobDistributor("output", 10, scorefxn)`	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
`jd.native_pose = native_pose`	Sets the native pose for RMSD comparisons
`jd.job_complete`	Returns True if all decoys have been output
`jd.output_decoy(pose)`	Outputs `pose` to a file and increments the decoy number

`while not jd.job_complete:`
`  # Code for creating decoys`
`  jd.output_decoy(pose)`
	Loop to create decoys until all have been output

`jd.additional_decoy_info = "Created by Andy"`	Sets a string to be output to the pdb files