Workshop #11: Custom Movers & Energy Methods

You may have noticed that every "mover" we have introduced in the preceding chapters has shared an `apply()` method, which takes a pose as its argument. One of the benefits of object-oriented programming is the ability to design a group of objects, such as movers, that are all related to each other. More complicated objects are said to inherit methods and/or data from simpler objects, known as base classes. Rosetta 3 has been carefully designed in an organized way with countless derived classes inheriting code from higher up in the inheritance tree.

Suppose you wish to create your own movers or score function scoring components for a specific project. PyRosetta allows you to make your own custom movers and energy methods that inherit from Rosetta's `Mover` and `EnergyMethod` base classes just as movers like `SmallMover` and `MinMover` do. This workshop will provide a quick overview of classes and inheritance in Python and demonstrate how to customize movers and energy methods.

Suggested Reading

1. Python help on classes available at http://docs.python.org/tutorial/classes.html
2. A. Leaver-Fay, *et al.*, "ROSETTA3: An object-oriented software suite for the simulation and design of macromolecules," *Methods Enzymol.* **487**, 545–574 (2011).

Classes in Python

Python is more than just a scripting language; it is designed for object-oriented programming, including class definitions, inheritance, instantiation, class properties, and class methods. A full description of object-oriented programming in Python will not be covered here, but we will provide a quick overview, using a simple example.

Let's say we wish to create two objects in Python that contain properties and methods for drawing shapes on the screen — one for drawing a circle and another for drawing a square. Since a circle and a square are both shapes, they share a lot in common. It is a good programming practice to first create a "base class" `MyShape`, which will contain properties and methods that `MyCircle` and `MySquare` will "inherit".

The `class` statement is used to declare a class; the properties and methods for the class are defined indented underneath the class declaration in the class statement block:

```
class MyShape:
    """A base class for a generic shape."""
    def __init__(self):
        self.color = "black"  # default value for color
```

The `__init__()` method of any class — known as the "constructor" — is called whenever a new object is instantiated as part of that class. If we were to create a new `MyShape()` object using `shape = MyShape()`, the code below `def __init__(self):` would be run.

Note the use of the variable name `self` in the method declaration for `__init__()`. The keyword `self` refers to the particular instance of `MyShape` that is running the code. Whenever one calls *any* class method in Python, the first argument for the method is always the instance of the class calling the method. (For example, when we type `pose.total_residue()`, we are running a function within the `Pose` class and passing the variable `pose` as the first argument of `total_residue()`.) In the example above, the variable `color` stored within whichever object called `__init__()` is set to the value `"black"`. Such a variable stored within an object is called a "property".

Another common method for a class is `__str__()`. The function `__str__()` returns a string version of the object; that is, it will code for what happens if one tries to `print` the object:

```
def __str__(self):
    return self.__doc__
```

The keyword `__doc__` is a predefined variable in Python that stores the value of the "docstring" of its object. Any block comments after a declaration between a pair of three double quotes (`"""`) become the docstring for that class or method. If one were to type `print shape`, Python would return, in this case, `A base class for a generic shape.`

For our example class, we could code for a method that returns the area of the shape:

```
def area(self):
    """Return the area of the shape."""
    return # Code to calculate the area goes here.
```

Because `MyShape` is a base class and every shape has a different method for calculating its area, we simply return nothing here. This is a way to inform other programmers (or ourselves) that any shape objects created later should include and fully implement this method for that particular shape.

We could also include a method for outputting the shape on the screen. We will provide a default value for this method also. This allows one to call either `MyShape.draw()` or `MyShape.draw(<some_integer>)`, and either one will work.

```
def draw(self, line_width=1):
    """Draw the shape on the screen."""
    pass  # Code to draw the shape goes here.
```

Now we will code for a derived class, `MyCircle`. In our declaration line for `MyCircle`, we will include in parentheses the base class `MyShape`. This will cause `MyCircle` to "inherit" from `MyShape` all of its methods so we do not have to code them again. However, we will recode the `__init__()` and `area()` methods, because those are unique for a circle.

```
class MyCircle(MyShape):
    """A subclass of MyShape for a circle."""
    def __init__(self):
        # This overrides the __init__() method inherited
        # from MyShape.
        MyShape.__init__(self)
        self.radius = 1.0  # default value

    def area(self):
        """Return the area of the circle."""
        # This overrides the area() method inherited from
        # MyShape.
        return math.pi * self.radius**2
```

Notice how we call the `__init__()` method of the base class `MyShape`. Doing such will set the `color` property of the instance of `MyCircle` when it is initialized or constructed. In this way we can inherit a class's method and then add additional things to it, such as the addition of the definition of the property `radius` here.

One could make a similar class `MySquare` with different `__init__()` and `area()` methods.

1. Create a single Python file (`my_shapes.py`) containing the above classes. In addition, create a class `MySquare` that initializes a property `side_length` and includes a mathematically appropriate `area()` method. Import the two derived classes into IPython using `from my_shapes import MyCircle, MySquare`, and then run the following lines of code:

   ```
   circle = MyCircle()
   square = MySquare()
   print circle
   print square
   print circle.color
   print square.color
   square.side_length = 2
   print square.area()
   circle.radius, circle.color = 1.5, "pink"
   print circle.area()
   circle.draw(2)
   ```

   After typing the above lines, what is the area of your square? ____ What is the area of your circle? _____ Assuming that you had put actual code for drawing a circle into the `draw()` method, what would the color of the drawn circle be? _____ What would be the line width of the drawn circle? ___

Custom Mover Classes

Now we have a foundation for creating our own mover classes in PyRosetta that are subclasses of a base class `Mover`.

2. Create a Python file with the following lines of code. (Be sure to `import rosetta` at the top of the file, but you do not need to call `init()`.)

```
class PhiNByXDegreesMover(rosetta.protocols.moves.Mover):
    """A mover that increments the phi angle of residue N
    by X degrees.

    Default values are residue 1 and 15 degrees.

    """
    def __init__(self, N_in=1, X_in=15):
        """Construct PhiNByXDegreesMover."""
        rosetta.protocols.moves.Mover.__init__(self)

        self.N = N_in
        self.X = X_in

    def __str__(self):
        return "residue: " + str(self.N) + \
               "  phi increment: " + str(self.X) + \
               " degrees"
```

To function as a true Rosetta mover, we must do a few specific things. First, we must make our class inherit from `rosetta.protocols.moves.Mover`, a Rosetta base class. Furthermore, in the initialization code for our new mover object, we must run the `__init__()` constructor method of the Mover base class. One can also inherit from other Rosetta movers besides `Mover`. If this is done, simply call that particular mover's constructor instead. (You can — and should — include other mover initializations, such as is done in the code above, *e.g.*, `self.N = N_in`.)

Now expand the above class with the following two methods:

```
    def get_name(self):
        """Return name of class."""
        return self.__class__.__name__

    def apply(self, pose):
        """Applies move to pose."""
        pose = pose.get()
        print "Incrementing phi of residue", self.N, "by",
        print self.X, "degrees...."
        pose.set_phi(self.N, pose.phi(self.N) + self.X)
```

All movers must include the above two methods, namely, `get_name()` and `apply()`. The `get_name()` method for *every* PyRosetta mover you make need only include the line `return self.__class__.__name__`, which will return the name of the class of the object, in this case, `PhiNByXDegreesMover`.

The `apply()` method contains your custom code that alters the pose. Note that you must first call the `get()` method on `pose` as shown above. (The reason why this must be done is related to the underlying C++ access pointers and is unfortunately beyond the scope of this workshop.)

3. Import the `PhiNByXDegreesMover` class from the Python file containing it. Construct two instances of the mover, one with `N` set to 1 and `X` set to 15, the other with `N` set to 10 and `X` set to 45. (Note that properties of movers made in PyRosetta can be accessed and set directly — as you did for the properties in our shapes example — whereas, those from the Rosetta3 library must be accessed and set using getter and setter methods.) Load a test pose and apply both of your movers. Confirm that your movers behaved as expected using PyMOL.

4. Create a mover that decrements the psi angle of every *even* residue in a pose by a value passed during initialization. Bonus: See if you can do this by using a *single instance* of `PhiNByXDegreesMover` called from within a `for` loop in your new mover.

5. Create a `TrialMover` containing `PhiNByXDegreesMover` and one containing a similar `PsiNByXDegreesMover`. Write a script that runs a Monte Carlo algorithm using these `TrialMover`s to fold a small protein.

Decorators in Python

Setting up a custom energy method with a corresponding scoring component is more complicated than creating a new mover. Fortunately, to assist us with this process, we can use a "decorator", which helps us to prepare all of the required class methods.

Decorators are a high-level coding feature available in Python. A decorator is a function that takes a class, object, or method as input and returns a modified, or decorated, version of that class, object, or method. For example, suppose we have a class, `MyCircle`. We could define a decorator function such as the following:

```
def hollow(shape_in):
    """Modify the draw() method of an input shape class to
    output a hollow shape."""
    # Code to modify the draw() method goes here.
    return shape_out
```

We can now pass an instance (an object) of `MyCircle` to our decorator to get a modified version of the object:

```
circle = MyCircle()
circle.draw()  # Draws a filled circle.
hollow_circle = hollow(circle)
hollow_circle.draw()  # Draws a hollow circle.
```

We can also pass the class itself to our decorator function. In this example, we will overwrite our old `MyCircle` class with the new, decorated, hollow one:

```
MyCircle = hollow(MyCircle)
hollow_circle = MyCircle()
```

Python provides an alternate "wrapper" syntax, in which the entire class definition block of code is passed to the decorator function:

```
@hollow
class MyCircle(MyShape):
    # The rest of the class definition would go here.
```

If the `MyCircle` class definition is decorated like this by `@hollow`, then every `MyCircle` object will have a "hollow" `draw()` method.

Custom Energy Methods

It is unlikely that you will need to write your own decorator functions for structure prediction or design applications. However, there is a decorator in the PyRosetta library already written for you that will modify any custom energy method classes you write so that your custom classes will contain most of the required methods automatically. For example, here is how we can create a custom context-independent, one-body scoring method that will score a pose based solely on the number of residues. We will call our new class `LengthScoreMethod`. First, we must import the proper parent class method from Rosetta:

```
from rosetta.core.scoring.methods import
                              ContextIndependentOneBodyEnergy
```

Now, we will define our class:

```
@rosetta.EnergyMethod()
class LengthScoreMethod(ContextIndependentOneBodyEnergy):
    """A scoring method that favors longer peptides by
    assigning negative one Rosetta energy unit per
    residue.

    """
    def __init__(self):
        """Construct LengthScoreMethod."""
        ContextIndependentOneBodyEnergy.__init__(self,
                                                 self.creator())

    def residue_energy(self, res, pose, emap):
        """Calculate energy of res of pose and set emap"""
        # 1 energy unit per residue
        emap.get().set(self.scoreType, -1.0)
```

And that's it! We should point out a few things. First, like when we inherited from `Mover`, we need to call the parent energy method class's `__init__()` constructor. An energy method constructor requires an additional argument, a "creator" function; where did this argument `self.creator()` come from? We did not explicitly define a method `creator()` to pass to `ContextIndependentOneBodyEnergy`'s constructor. The decorator, `EnergyMethod()`, (which is a callable class just like `ScoreFunction()`,) does that for us. `EnergyMethod()` also makes life easier by defining several other required methods for us.

Before we explain how this new energy method works, let's demonstrate how we can use it.

6. Create a Python file with the `LengthScoreMethod` code. (Include the proper imports.) Import Rosetta, import the new energy method, create a pose from the sequence "ACDEFGHIKLMNPQRSTVWY", and construct an empty `ScoreFunction()` with `sf = ScoreFunction()`. What is the score of your pose? ____

   Create a variable to hold the score type for your custom energy:

   ```
   len_score = LengthScoreMethod.scoreType
   ```

   (Note that we do not have to instantiate a `LengthScoreMethod` object; we simply can extract the score type directly from the class.)

   Now set the weight for the `len_score` scoring component to 1, just as we would for any other scoring component, such as `fa_atr`:

   ```
   sf.set_weight(len_score, 1.0)
   ```

   Score the pose. What is the score for the pose now? ____

Let's return to the code for our new energy method to understand how it works.

For one-body energy methods, whenever one scores a pose, the `ScoreFunction()` loops through each residue in the pose and calls the `residue_energy()` method of the energy method class associated with each score type. The `residue_energy()` method uses the residue and pose objects passed as arguments to calculate an energy score and sets the corresponding score type in the passed energy map. In our example above, we assign -1 energy unit per residue, independent of the context (*e.g.*, what type of residue it is), so we did not need to do anything with the arguments `res` or `pose`; we simply had to set `self.scoreType` in `emap` to `-1.0`.

Note that, as with custom mover `apply()` methods, because the `residue_energy()` method is called by a Rosetta function (written in C++, not a Python), we need to "`get()`" `emap` or any other passed object before we can use it.

7. Create an energy method that gives a favorable (-1.0) score for each aromatic residue in a pose. (Remember to call `res.get()`.) Repeat programming exercise 3c and d of Workshop 6, except add your aromatic-favoring energy method score type to the full atom standard score function to bias the design. How do your results change? How much do you have to adapt the weight of your score type before all of the designed residues are aromatic?

The procedure for making a custom two-body energy method is similar, but there are a few additional required methods and `residue_energy()` is replaced by `residue_pair_energy()`. Below is a template one can use for creating a context-independent, two body score type:

```
from rosetta.core.scoring.methods import
                            ContextIndependentTwoBodyEnergy


@rosetta.EnergyMethod()
class CI2BScoreMethod(ContextIndependentTwoBodyEnergy):
    """A scoring method that depends on pairs of residues.
    """
    def __init__(self):
        """Construct CI2BScoreMethod."""
        ContextIndependentTwoBodyEnergy.__init__(self,
                                            self.creator())

    def residue_pair_energy(self, res1, res2, pose, sf,
                            emap):
        """Calculate energy of each pair of res1 and res2
        of pose and set emap."""
        # A real method would calculate a value for score
        # from res1 and res2.
        score = 1.0
        emap.get().set(self.scoreType, score)

    def atomic_interaction_cutoff(self):
        """Get the cutoff."""
        return 0.0  # Change this value to set the cutoff.

    def defines_intrares_energy(self, weights):
        """Return True if intra-residue energy is
        Defined."""
        return True

    def eval_intrares_energy(self, res, pose, sf, emap):
        """Calculate intra-residue energy if defined."""
        pass
```

The template for a context-dependent, two-body method is identical, except that it inherits from and initializes `ContextDependentTwoBodyEnergy` instead.

Programming Exercises

1. Pick a small protein and use the `toolbox` method `get_secstruct(pose)` to determine the secondary structure, store that information in the pose, and output the results. Using the method `Pose.sectruct(resnum)`, and `PhiNByXDegreesMover`, loop through all residues that are a part of loops, set `X` between 1 and 15 stepping by 1

each time, apply for each case, and record the *change* in score. Reset the pose after each move. Plot the average change in score vs. `X`. Repeat this process for helix and strand residues, and then do the same three plots using `PsiNByXDegreesMover` instead. Do the sets of plots look similar for phi and psi? Does secondary structure appear to have an effect on how large `X` can be before the score is severely penalized? Do the default `angle_max` values of 0°, 5°, and 6° for helix, sheet, and loop, respectively, make sense based on your plots?

2. Create a context-dependent, two-body energy method to reward structures containing salt bridges. The energy method should give a bonus for each pair of acid and base side chains within 6 Å of each other. Find a small protein with several salt bridges. Run a *de novo* folding algorithm with and without your new scoring method. Does scoring for salt bridges give better results, that is, are the RMSDs from the native structure lower?