Lab #8

Physics 91SI, Spring 2013

Objective: In this lab, you will be introduced to object-oriented programming in Python and will get experience writing classes, along with importing and inheriting.

As usual, log on to `corn` using `ssh -X` and clone over the starter repository:

```
hg clone /afs/ir.stanford.edu/class/physics91si/src/lab8 lab8
```

Remember to `hg commit` often to save your changes, and **submit your code at the end of the lab.**

Part 1: Write a **Set** Class

A couple of weeks ago, when we discussed Abstract Data Types, we mentioned Sets and what they are and how they behave. As a quick recap, a Set is a collection of arbitrary objects, with the caveat that no object is repeated, i.e. it can show up more than once in the Set. From this, we have the usual collection methods, like adding an object, removing an object, and asking for the size of the collection, but Sets also have special operations called union, intersection, and set difference.

The union of two Sets, which we'll notate `SetA | SetB`, returns a new Set with all of the members that were either in SetA or SetB or both. The intersection of two Sets, which we'll notate `SetA & SetB`, returns a new Set with the members that were in both SetA and SetB. The difference of two Sets, or `SetA - SetB`, returns a new Set with all of the members that were in SetA but not in SetB.

In the `lab8` folder, there is a file called `sets.py` which has the skeleton of a Set class definition. Edit that file and fill in the code for all of the methods that contain `pass`. Remember that you can add any attributes that you think are necessary for implementing the class and you should write short docstrings for your methods that explain what they do, what the arguments should be, and what is returned.

After the normal methods in the `sets.py` file, there are skeleton definitions for operator overloading methods (which look like `__or__` , with two underscores at the beginning and two more at the end). When you're writing a new class, you can implement as many or as few of these as you want. The ones we outlined in the file allow you to use built-in functions like `len()` and built-in operators like `|` and `&` on Set objects. Once each of these method definitions is completed, each should `return` the expected output of the operation (as outlined above).

Also, if you're interested, check out the code for the Set class iterator at the very end of the file. The `__iter__()` and `next()` methods define how to for loop over a Set object.

Once you've implemented all of the methods for the Set class, you should write a test script in `settest.py` that makes sure your methods work properly. Your test script includes

```
from sets import Set
```

and then should do some things like add and remove a bunch of values and then check to see if `MySet.size()` returns the proper size. Even better, test your union, intersection, and subtraction methods, by doing something like checking the intersection and difference of a Set of odd numbers and a Set of prime numbers (which you can generate yourself by just having a few manual `.add()` calls).

Part 2: Using and Inheriting other Classes into a New Class

Now that you've written a Set class, you're going to write a new class that will inherit your Set class as a parent class. You will also be using the Student class from lecture today as part of your implementation, which has been provided in the `lab8` folder.

In your `lab8` folder, you should find a file called `classlists.py`. You'll be writing a new ClassList class, which is a Set of all Students in a class (this last time, class means like a school class, sorry). ClassList will inherit Set, but it will use the Student class in the implementation, and the difference in roles is important to understand. ClassList will inherit all of the method and data attributes defined in the Set class, but you will need to overwrite the add, remove, and contains methods in your ClassList definition because you'll want them to work differently.

You must redefine `add`, `remove`, and `contains` so that the user can simply send a string name as an argument, and then in the implementation, internally, your ClassList methods will create a new Student object, with that string name given as the argument, and then will add/remove/check for that Student object in the `self.member` list. You'll find an == operator overloading method in `students.py`, which defines that `StudentA == StudentB` will return True if their name strings are equivalent, so use this to compare Student objects when implementing these method re-definitions.

As a challenge, there is skeleton code at the bottom of `classlists.py` for re-defining the iterator to give the Student names in alphabetical order, and for overloading the slice operator, e.g. `MyClassList['A':'H']`. Try implementing these methods if you are feeling adventurous!