Lab #3

Physics 91SI Spring 2012

Objective: This lab will introduce you to the basics of the Python programming language, using short programs that you will run from the command line and in the interpreter.

As usual, log in to corn.stanford.edu, with X-forwarding enabled (ssh -X). We've prepared some starter code for this lab, which has the required boilerplate "skeleton" of a python program, and implements a couple of the trickier functions. To get started, go to your physics91si directory and clone the starter repository:

```
hg clone /afs/ir.stanford.edu/class/physics91SI/src/lab3 lab3
```

Remember to hg commit often to save your changes!

Part 1: Fibonacci Sequence

This is probably the second program you ever wrote after "Hello World!", so why not try it in Python. The Fibonacci sequence is defined as $f_{n+1} = f_n + f_{n-1}$, giving the sequence:

```
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 ...
```

The ratio of successive elements converges to $\phi = (1+\sqrt{5})/2 \simeq 1.61803$, the famous "golden ratio." Write a program that computes the first *n* elements of the Fibonacci sequence and prints them to the terminal, one per line. Finally, have your program print out the ratio of the last two elements $f_n / f_{n-1} \simeq \phi$.

Your program should take one command-line argument, a positive integer representing the number of elements to compute; the starter code in fib.py has already been written to handle this. You need only fill in the rest of the main() function - this should take less than 15 lines of code! Test your program by running it as usual from the command line, either:

```
python fib.py n   or    ./fib.py n
```

Part 2: Calculating Pi

Now for something a little more interesting. In the last problem, we calculated ϕ - now we'll calculate π using what's known as a "Monte Carlo" method. Instead of calculating π analytically from the limit of a series or an integral, the Monte Carlo method finds a fast approximation using random numbers and, in this case, the area of a circle. Here's one way to do it:

1. Take a large number of points (x,y), where x, y ϵ [0,1]
2. Count the number of points that fall inside the unit circle in this quadrant, i.e. the points where $x^2 + y^2 \leq 1$
3. Divide this by the total number of points to approximate the area of this segment = $\pi / 4$

In pi-monte.py, write a python program that takes, as before, the number of points to use, and prints out an approximation to π. To generate a random number between 0 and 1, use x = np.random.random() - we'll talk more about this numpy package in Week 4. When you are confident in your implementation, try running with a large N, and see how close you can get!

Part 3: Modules / Challenge

While the above exercises show you how to use Python as any programing language, when running data analysis and simulations, often you'll want to work with Python in "interactive mode" through the interpreter. In your lab3 directory, open python and type import analysis. Now you can access the functions in analysis.py just as in a normal program. As a bonus, Python has a handy auto-documentation feature - just type help(analysis) for a list of available functions.

If you look at analysis.py, you'll notice that most of the functions operate on and return "*data*," which is a list of tuples (x, y) that represent data points. (We'll talk more about these types on Thursday, but for now you can treat them like arrays in Java or C++.) Working in the interpreter, load the data from data1.dat and plot it using the included functions.

You have three tasks here:

1. Write the max_y_index() function, which iterates over *data* and returns the index of the maximum y value (this is more useful than the built-in max(), which just gives us the value).
2. Next, make a version of this called find_peak() which takes takes *data* and two floats, xmin and xmax, and returns the index of the maximum y value on that interval. Your function should work even if xmin and xmax don't match the data points exactly! (*Hint: an elegant solution involves helper functions and making sure the data is sorted, but there's a very easy way if you've done #1*)
3. With these functions written, use the included plot() and label() functions to make annotated plots of the points in data1.dat and data2.dat. You can save each by clicking the save icon that appears in the plot window. Save each plot as a .png image, and use hg add and hg commit to add them to your repository.

Part 4: Submitting your code

From now through the rest of the course, we'll be using Mercurial to submit code. We've already set up submission repositories for each lab in the course directory. All you need to do now is "push" your work to this repository. cd to the lab directory (you need to be inside it), commit any changes, and use the following command:

```
hg push /afs/ir.stanford.edu/class/physics91SI/submissions/yourname/lab3
```

or

```
hg push $SICDR/submissions/yourname/lab3 (if you set the environment
                                variable)
```

If you got through Part 3 of this lab, be sure that you've added the images to your repository before you push. (You can push multiple times, if you need to.)

Also, if you didn't get through the last part of Lab #2, cd to that directory and submit it as above (to $SICDR/yourname/lab2, of course).