Lab #3

Physics 91SI Spring 2013

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. Today we'll explore both of the major applications of scientific computing: **computer modeling** and **data analysis**.

Parts 1 and 3 help you write programs to calculate and display approximations to two common mathematical constants, φ (the Golden Ratio) and pi. These programs serve as an introduction to the techniques used in computer modeling.

In **part 2**, you will explore the data analysis side of computing. You will also practice using Python's interactive mode, as well as producing plots of your results.

In **part 4**, you will submit your code for the day.

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 "skeleton" of a Python program and implements a couple of the trickier functions. In parts 1–3, you'll open the starter files and write the code for the functions that have been left blank. 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: The Fibonacci Sequence and Calculating the Golden Ratio

The Fibonacci sequence is defined by $f_{n+1} = f_n + f_{n-1}$, 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 $\varphi = (1 + \sqrt{5})/2 \approx 1.61803$, the famous "golden ratio."

Your program will take one command-line argument *n*, a positive integer representing the number of elements to compute; the starter code has already been written to handle this. Your program should compute the first *n* elements of the Fibonacci sequence and print them to the terminal, one per line. Afterwards, your program should print out the ratio of the last two elements $f_n/f_{n-1} \approx \varphi$.

Go ahead and open the starter code in `fib.py`. In it, write code in the the `main()` function to accomplish the above tasks. Test your program by choosing a number and running it like a UNIX command from the command line, either:

```
python fib.py <number>   or  ./fib.py <number>
```

Part 2: Modules: Data Analysis and Plotting

From within your `lab3` directory, run `python` and type `import analysis`. Now you can now access the functions in `analysis.py` by typing e.g. `analysis.load()`. You can type `help(analysis)` for a list of available functions. (We have documented our code in such a way that Python lets you access it through this command.)

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 this like an array of length-2 arrays in Java or C++.) Working in the interpreter, load the data from the file `data1.dat` and plot it using the functions in `analysis.py`.

After plotting, quit the interpreter. Now analyze the data by completing these tasks:

1. Open `analysis.py` in a text editor and notice that the `max_y_index()` function is left blank. In it, write code to iterate over *data* and return the index of the maximum y value (this is more useful than the built-in `max()`, which just gives us the value).
2. Note that the `find_peak()` function is also blank. In it, write code to take *data* and two floats, `xmin` and `xmax`, and return 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.
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 3: 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, a 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 $x, y \in [0,1]$.

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 $= \pi/4$.

Open `pi-monte.py`. In it, 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 4: Submitting your code

As always, you'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) and commit any changes. Then use the following command:

```
hg push /afs/ir.stanford.edu/class/physics91si/submissions/yoursuid/lab3
```

or

`hg push $SICDIR/submissions/yoursuid/lab3` (if you set the environment variable in lab 2)