ECON-UB233_hw0

September 1, 2014

0.1 Lab Report #0: Python Practice

Revised: August 18, 2014

This should be completed before the second class, but you can easily do it before the term starts. It will not be collected or graded, but you should do it anyway. Seriously. You'll regret it later if you don't.

Python is a popular high-level computer language, which means it's easier to use than C++ or Java and (far) more powerful than Excel. It's widely used in quantitative segments of the business world, including consulting, banking, and asset management.

What follows will get you started. It should take you about an hour.

0.1.1 1. Accessing Python.

You can access Python in a 3 ways:

- wakari.io: A zero-setup (except for creating a *free* account) cloud based scientific python system online.
- On your machine. For instructions see quant-econ.net. If you choose this route you should follow the steps there to install the Anaconda Python distribution unless you have prior Python experience.

Choose one of the above and launch an IPython shell (enter the command `ipython` from the windows command propt or unix terminal) and type the following commands (you can copy/paste if you'd like – either way we will need them later on):

```
In []: import numpy as np
       import matplotlib.pyplot as plt
       import sympy as sym

       # if you are working in the python notebook enter the command below
       # otherwise, don't
       %matplotlib inline
```

0.1.2 2 . Scalar Operations

We'll start by entering commands on the in the IPython interpreter. If you can't find it, look for `In [1]:` (the "prompt").

(a) Type each at the prompt

```
In []: x=7
```

```
In []: x = 7
```

```
In []: x
```

```
In []: x/2
```

```
In []: 7**2
```

What do each of these do?

(b) Now type these:

```
In []: np.pi
```

```
In []: np.exp(1)
```

```
In []: e, x = np.exp(2), np.log(x)
```

What do they do?

0.1.3 3. Vector Operations

NumPy arrays do arithmatic and apply many functions elementwise. What this means is that you can do a bunch of things at once rather than copying the same command over and over again as you would in (say) Excel. Here's an example:

(a) Generate a grid with the command:

```
In []: x = np.linspace(-3, 3, 13)
```

What does `x` look like? (Type `x` at the propt to find out.)

(b) Now compute x^2 for all values of `x`:

```
In []: xsquared1 = x**2
       xsquared2 = x*x

       xsquared1, xsquared2
```

These two lines do the same thing, namely square each element of x.

(c) Plot the results this way:

```
In []: plt.plot(x, xsquared1, "b")
       plt.plot(x, xsquared2, "r--")
```

Type `plot?` to learn how to control the type and color of lines. Can you make the line red? Magenta? Dashed? Can you replace the line with circles?

(d) Using the same `x`, plot the functions `st.norm.pdf(x)` and `st.norm.cdf(x)`. (You can enter these expressions directly in the plot command.) Use the `?` magic command (a technical term from IPython) to find out what these functions are — for example, `st.norm.pdf?`. What do you see? How would you make the line smoother?

0.2 4. Vector Operations (continued)

We can also work with data vectors. Suppose, for example, we have five observations of some variable x defined with the command `x = np.array([1, 7, -2, 4, 8])`.

Explain each of the following: `np.mean(x)`, `sum(x)`, `np.sum(x) len(x)`, `x.size`, `x.shape`, `np.size(x)` and `sum(x)/len(x)`. As we noted earlier, you can get more information about these commands using the `?` magic command at the IPython prompt. You can also Google "Python command" (or "Python *package name* command) or view the documentation for the package being used (try Googling"numpy documentation").

0.2.1 5. Symbolic Math

There are lots of computer programs that do symbolic math: solve equations, differentiate functions, and so on. We'll use Python to derive the moments of the normal distribution from its moment generating function. You'll use similar commands in future assignments, so pay attention.

Type these commands in the command line:

```
In []: mu, sigma, s = sym.symbols("mu, sigma, s")
       mgf = sym.exp(mu*s + (sigma*s)**2/2)
       mu1 = mgf.diff(s, 1).subs({"s": 0})
       mu2 = mgf.diff(s, 2).subs({"s": 0})
       mu4 = mgf.diff(s, s, s, s).subs({"s": 0, "mu": 0, "sigma": 1})

       (mu1, mu2, mu4)
```

What do they do? What happens if you skip the first line? What is `mgf`? What is `mgf.diff(s,2)`? How would you use `?` to find out more about `diff`? What does `subs` do?

0.2.2 6. Data input and scripts.

Our last task is to input data from a spreadsheet using a "script": that is, we put the commands in a file so we can run them over again without typing everything in from scratch.

(a) Enter the following into a spreadsheet

$x1$	$x2$
1	2
3	4
5	6

Save it as an Excel workbook with name (say) `testdata.xls`. Make sure you know what directory it's in.

Comments: (i) You can use any file name you like, but the same name must be used in the script below. (ii) You need to do something to tell Python what directory the file is in. How that works depends on what version you're using.

(b) We're going to wite a Python script to read the contents of the spreadsheet into Python. Create a new file in your text editor and enter the following commands:

```
In []: """
       Practice script for data input from spreadsheet this is in a docstring
       """
       import pandas as pd
       print("Spreadsheet input")

       # Read in the data -- this is a comment
       data = pd.read_excel("testdata.xlsx")
```

Save these commands under some appropriate file name (something ending in `.py`) in the same directory as the spreadsheet.

Now run this program within IPython by starting an IPython process in this same directory and entering `run FILENAME.py`, where `FILENAME` is the name you just chose to save those commands under. To see if works, type `data` at the prompt and press enter. If it does, great! If not, welcome to the world of programming, where most of your time is spent fixing bugs. Remind yourself that patience is a virtue. And speak to others about your problem, that often helps even if they don't have the answer.

(c) Add to your program a line that plots x1 against x2 and marks each data point with a circle.

0.2.3 7. Functions

Python allows you to write functions in a couple ways.

(a) The most common way is to write the function in a `.py` file using the `def` keyword. Here is an example of what that might look like

```
In []: def helloworld(x):
           """
           Comments

           If you type helloworld? you'll get these comments

           The program prints "Hello World!" and the value of the variable `x`

           Examples
           --------
           >>> helloworld(7)
           Hello World!
           7
           >>> helloworld(np.arange(1, 8))
           Hello World!
           [1 2 3 4 5 6 7]
           """
           print("Hello World!")
           print(x)
           return
```

(b) A more compact way to write a short function is with an "anonymous function." The terminology is obscure, but here's how it works. Type the commands:

```
In []: myfunction = lambda x: x**2
       myfunction(7)
```

You should get 49. We'll use something like this when we get to option pricing, where we need to use a long formula repeatedly. More on anonymous functions in the Python documentation or by googling `lambda functions python`.