Topics in Computational Economics

Lecture 3

John Stachurski

NYU 2016

Today's Lecture

- UNIX shell – more features
- Hardware and execution
- Introduction to Python

The Desktop and the UNIX Shell

Getting started with the shell:

- See `command_line.ipynb`

Further reading

- http://swcarpentry.github.io/shell-novice/

Exercise:

Clone `https://github.com/jstac/tmp_repo`

Using just **one line** in the shell, delete from the `files` directory all files that do **not** end in `.py`

Hints:

- Google is your friend
- Make sure you're in the right dir before deleting!
- If you mess up, `git reset --hard` restores files

Python!

Hopefully you did the homework:

- http://quant-econ.net/py/getting_started.html

Please start up jupyter notebook

- consider adding `alias jp='jupyter notebook'`

Questions about the modal interface? Anything else?

An Easy Python Program

Next step: write and pick apart small Python program

Objective: To simulate and plot

$$\epsilon_0, \epsilon_1, \ldots, \epsilon_T \quad \text{where} \quad \{\epsilon_t\} \overset{\text{IID}}{\sim} N(0,1)$$

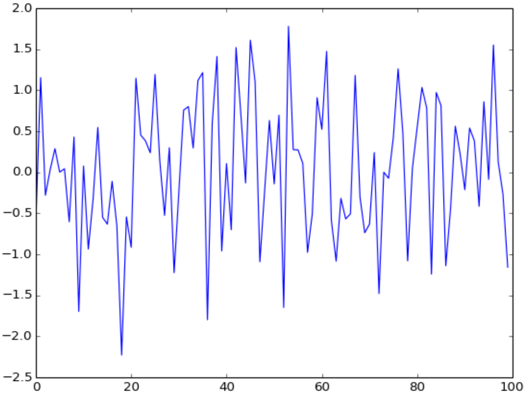
Source:

http://quant-econ.net/py/python_by_example.html

Like all first programs, it **will be contrived**

(Avoid existing solutions to focus on Python syntax)

The end result should be this (modulo randomness)



A first pass:

```
import matplotlib.pyplot as plt
from random import normalvariate

ts_length = 100
epsilon_values = []    # An empty list
for i in range(ts_length):
    e = normalvariate(0, 1)
    epsilon_values.append(e)
plt.plot(epsilon_values, 'b-')
plt.show()
```

Import Statements

First, consider the lines

```
import matplotlib.pyplot as plt
from random import normalvariate
```

Here matplotlib and random are two separate **modules**

- module = file containing Python code

Importing a module makes the functionality in the module available to the user

Example. Importing the module `random` causes Python to

- run the code in library file `random.py`
- initializes variables, functions, etc. in the file
- provides access to them via `random.name_of_object`

```
>>> import random
>>> random.normalvariate(0, 1)
-0.12451500570438317
>>> random.uniform(-1, 1)
0.3512161619700336
```

You can also just import the names directly, like so

```
>>> from random import normalvariate, uniform
>>> normalvariate(0, 1)
-0.38430990243287594
>>> uniform(-1, 1)
0.5492316853602877
```

Lists

Statement `epsilon_values = []` creates an empty list

Lists: a Python data structure used to group objects

```
>>> x = [10, 'foo', False]
>>> type(x)
<type 'list'>
```

Note that different types of objects can be combined in a single list

Adding a value to a list: `list_name.append(some_value)`

```
>>> x
[10, 'foo', False]
>>> x.append(2.5)
>>> x
[10, 'foo', False, 2.5]
```

- `append()` is an example of a **method**
- method = a function "attached to" an object

More examples of list methods:

```
>>> x
[10, 'foo', False, 2.5]
>>> x.pop()
2.5
>>> x
[10, 'foo', False]
>>> x.reverse()
>>> x
[False, 'foo', 10]
```

In general, different types of objects have different methods

- list objects have list methods
- string objects have string methods, etc.

Examples of string methods:

```
>>> s = 'foobar'
>>> s.endswith('bar')
True
>>> s.upper()
'FOOBAR'
```

Warning: Like C, Java, etc., lists in Python are zero based

```
>>> x
[10, 'foo', False]
>>> x[0]
10
>>> x[1]
'foo'
```

The for Loop

Consider again these lines from `test_program_1.py`

```
5  epsilon_values = []    # An empty list
6  for i in range(ts_length):
7      e = normalvariate(0, 1)
8      epsilon_values.append(e)
```

Lines 7–8 are the **code block** of the `for` loop

Reduced indentation signals lower limit of the code block

Comments on Indentation

In Python **all** code blocks are delimited by indentation

- more consistency, less clutter

Notes:

- Line before start of code block always ends with :
- All lines in a code block must have same indentation
- The Python standard is 4 spaces—please use it
- Tabs and spaces are different
  - But Jupyter will add 4 spaces when you hit Tab
  - So will Vim with the vimrc I gave you

While Loops

Here's the same program with a while loop (test_program_2.py)

```
import matplotlib.pyplot as plt
from random import normalvariate
ts_length = 100
epsilon_values = []
i = 0
while i < ts_length:
    e = normalvariate(0, 1)
    epsilon_values.append(e)
    i = i + 1
plt.plot(epsilon_values, 'b-')
plt.show()
```

User-Defined Functions

Now let's go back to the for loop

—but restructure our program to illustrate functions

To this end, we will break our program into two parts:

1. A **user-defined function** that generates a list of random variables
2. The main part of the program, which
   1. calls this function to get data
   2. plots the data

test_program_3.py

```python
import matplotlib.pyplot as plt
from random import normalvariate

def generate_data(n):
    epsilon_values = []
    for i in range(n):
        e = normalvariate(0, 1)
        epsilon_values.append(e)
    return epsilon_values

data = generate_data(100)
plt.plot(data, 'b-')
plt.show()
```

Our function `generate_data()` is rather limited

Let's make it more flexible by giving it the ability to return either

- standard normals, or
- uniform rvs on $(0, 1)$

This is done in `test_program_4.py`

```python
import matplotlib.pyplot as plt
from random import normalvariate, uniform

def generate_data(n, generator_type):
    epsilon_values = []
    for i in range(n):
        if generator_type == 'U':
            e = uniform(0, 1)
        else:
            e = normalvariate(0, 1)
        epsilon_values.append(e)
    return epsilon_values

data = generate_data(100, 'U')
plt.plot(data, 'b-')
plt.show()
```

In fact we can get rid of the conditionals all together

Method: pass the desired generator type *as a function*

To understand this, consider `test_program_6.py`

```python
import matplotlib.pyplot as plt
from random import normalvariate, uniform

def generate_data(n, generator_type):
    epsilon_values = []
    for i in range(n):
        e = generator_type(0, 1)
        epsilon_values.append(e)
    return epsilon_values

data = generate_data(100, uniform)
plt.plot(data, 'b-')
plt.show()
```

We can even go a step further and pass in all arguments to the function

```
import matplotlib.pyplot as plt
from random import normalvariate, uniform

def generate_data(n, generator_type, *args):
    epsilon_values = []
    for i in range(n):
        e = generator_type(*args)
        epsilon_values.append(e)
    return epsilon_values

data = generate_data(100, uniform, 0, 1)
plt.plot(data, 'b-')
plt.show()
```

Using the Scientific Libraries

In fact the scientific libraries will do all this more efficiently

For example, try

```
>>> from numpy.random import randn
>>> epsilon_values = randn(3)
>>> epsilon_values
array([-0.15591709, -0.67383208, -0.45932047])
```

Exercise

Simulate and plot the correlated time series

$$x_{t+1} = \alpha \, x_t + \epsilon_{t+1} \quad \text{where} \quad x_0 = 0 \quad \text{and} \quad t = 0, \ldots, T$$

Here $\{\epsilon_t\} \overset{\text{IID}}{\sim} N(0,1)$

In your solution, restrict your import statements to

```python
from random import normalvariate
import matplotlib.pyplot as plt
```

Set $T = 200$ and $\alpha = 0.9$

Solution

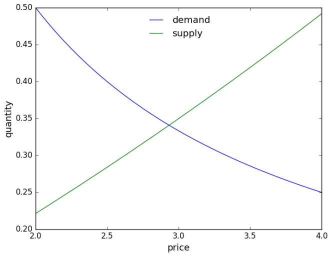
```
import matplotlib.pyplot as plt
from random import normalvariate

alpha = 0.9
ts_length = 200
x = 0

x_values = []
for i in range(ts_length):
    x_values.append(x)
    x = alpha * x + normalvariate(0, 1)
plt.plot(x_values, 'b-')
plt.show()
```

Homework

Computing equilibrium prices and quantities



See

https://github.com/jstac/quantecon_nyu_2016/tree/master/homework_assignments/hw_set2