Python in Economics and Finance

Part 2

John Stachurski, ANU

June 2014

Topics

- Data types
- OOP
- Iteration
- Functions
- NumPy / SciPy
- Matplotlib

Data Types

We have already met several native Python data types

```
>>> s = 'foo'
>>> type(s)
<type 'str'>
>>> y = 100
>>> type(y)
<type 'int'>
>>> x = 0.1
>>> type(x)
<type 'float'>
```

Some data types contain other objects:

- lists
- tuples
- dictionaries

These are called **container types**

Tuples are similar to lists

```
>>> x = ['a', 'b']  # Square brackets for lists
>>> x = ('a', 'b')  # Round brackets for tuples
>>> x = 'a', 'b'    # Or no brackets at all
>>> type(x)
<type 'tuple'>
```

In fact tuples are ”immutable” lists

Immutable means internal state cannot be altered

```
»> x = (1, 2)  # Tuples are immutable
»> x[0] = 10
Traceback (most recent call last):
  File ”<stdin>”, line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

Tuple unpacking:

```
>>> from scipy.stats import norm
>>> from scipy.integrate import quad
>>> phi = norm()
>>> value, error = quad(phi.pdf, -2, 2)  # Returns tuple
>>> value
0.9544997361036417
```

Dictionaries are similar to lists

Except items are named instead of numbered

```
>>> d = {'name': 'Frodo', 'age': 33}
>>> type(d)
<type 'dict'>
>>> d['age']
33
```

Of course there are also many third party types

```
>> import numpy as np
>> a = np.random.randn(4)
>> type(a)
<type 'numpy.ndarray'>
>> a.min()
-1.6403156232744336
```

Iterating

Step through sequence of data, performing given action

Python provides a nice interface to iteration via the `for` loop

Example of a for loop

```
animals = ['dog', 'cat', 'bird']
for animal in animals:
    print "The plural of " + animal + " is " + animal + "s"
```

Output:

```
The plural of dog is dogs
The plural of cat is cats
The plural of bird is birds
```

How does it work?

Example: The file us_cities.txt looks as follows

```
new york: 8244910
los angeles: 3819702
chicago: 2707120
houston: 2145146
philadelphia: 1536471
phoenix: 1469471
san antonio: 1359758
san diego: 1326179
dallas: 1223229
```

We want to clean it up like so:

```
New York        8,244,910
Los Angeles     3,819,702
Chicago         2,707,120
Houston         2,145,146
Philadelphia    1,536,471
Phoenix         1,469,471
San Antonio     1,359,758
San Diego       1,326,179
Dallas          1,223,229
```

Solution (filename = us_cities.py)

```
data_file = open('us_cities.txt', 'r')
for line in data_file:
    city, population = line.split(':')
    city = city.title()
    population = '{0:,}'.format(int(population))
    print(city.ljust(15) + population)
data_file.close()
```

Functions

Some are built-in:

```
>>> max(19, 20)
20
>>> type(max)
<type 'builtin_function_or_method'>
```

Others are imported:

```
from math import sqrt
```

We can also write our own functions

```
def f(x):
    return(x + 42)
```

One line functions using the lambda keyword:

```
f = lambda x: x**3
```

A common use of lambda

To calculate $\int_0^2 x^3 dx$ we can use SciPy's quad function

Syntax is quad(f, a, b) where

- f is a function and
- a and b are numbers

```
>> from scipy.integrate import quad
>> quad(lambda x: x**3, 0, 2)
(4.0, 4.440892098500626e-14)
```

Python functions are flexible:

- Any number of functions can be defined in a given file
- Any object can be passed to a function as an argument
- Functions can be defined inside other functions
- A function can return any kind of object, including functions

Object Oriented Programming

Traditional programming paradigm is called **procedural**

- A program has state (values of its variables)
- Functions are called to act on this state
- Data is passed around via function calls

In **OOP**, data and functions bundled together into **objects**

These bundled functions are called **methods**

Example: Lists = list data + list methods

```
>>> x = [1, 5, 4]
>>> x.append(7)
>>> x
[1, 5, 4, 7]
```

```python
from envelopes import Envelope

envelope = Envelope(
    from_addr='from@example.com',
    to_addr='to@example.com',
    subject='Envelopes demo',
    text_body="I'm a helicopter!")

envelope.add_attachment('/Users/bilbo/helicopter.jpg')
envelope.send('smtp.googlemail.com',
              login='from@example.com',
              password='password',
              tls=True)
```

Why OOP?

Fits human experience — many entities combine data and actions

```
class Agent:

    data:
        asset portfolio
        current labor shock

    methods:
        work
        trade
        consume
```

Use of *methods* also reduces global names

```
>>> x = ['foo', 'bar']
>>> x.append('fee')
```

The alternative would be something like `append(x, 'fee')`

But then you need another function in the global namespace

Python and OOP

Python is partly object oriented

- Everything is an object
- Native data types have methods
- Easy to build new objects bundling data and methods

But Python is not exclusively object oriented

Example: `x.append(val)` but `max(x)`

Everything is an object

In Python, everything is an object

```
>>> x = 1
>>> dir(x)
['__abs__',
 '__add__',
 .
 .
 'denominator',
 'imag',
 'numerator',
 'real']
```

Each Python object has a type, id, value, zero or more methods

```
>>> x = 1
>>> type(x)
<type 'int'>
>>> id(x)
10325096
>>> x.__add__(1)
2
```

Note that type is important in Python

```
>>> 10 + '10'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

We say that Python is **strongly typed**

Building New Classes of Objects

Example: Let's build a class to represent dice

Here's the pseudocode

```
class Dice:

    data:
        current_face -- the side facing up

    methods:
        roll -- roll the dice
```

Here's how the actual code looks—see file dice.py

```
import random

class Dice:

    faces = (1, 2, 3, 4, 5, 6)

    def __init__(self):
       self.current_face = 1

    def roll(self):
        self.current_face = random.choice(Dice.faces)
```

After running this code

```
»> d1 = Dice()
»> d2 = Dice()
»> d1.current_face, d2.current_face
(1, 1)
»> d1.roll()
»> d2.roll()
»> d1.current_face, d2.current_face
(6, 3)
»> d1.faces == d2.faces
True
»> d1.faces is d2.faces
True
```

Names and Namespaces

Names are symbols bound to objects in memory

The process of binding name to object takes place via assignment

```
x = 42
s = 'foo'
```

Python is **dynamically typed** — names are not specific to type

```
s = 'foo'
s = 42
```

A common problem in larger programs is name conflicts

For example

```
>>> from sys import *  # Import everything
>>> path
['/home/john/bin', ...]
>>> from os import *
>>> path
<module 'posixpath' from '/usr/lib/python2.7/posixpath.pyc'>
```

path from sys now shadowed by path from os

Namespaces

Python addresses this problem using namespaces

A **namespace** is a **mapping** from names to Python objects

Python uses multiple namespaces to give names context

For example, modules have their own namespace

```
>>> import sys
>>> sys.path
['/home/john/bin', ...]
>>> import os
>>> os.path
<module 'posixpath' from '/usr/lib/python2.7/posixpath.pyc'>
```

Class instances have their own namespace

```
>>> class Foo:
...     pass
...
>>> f = Foo()
>>> f
<__main__.Foo instance at 0x7f99db74f878>
>>> f.x = 42
>>> f.__dict__
{'x': 42}
```

When functions are invoked they get their own namespace

```
def f(x):
    a = 2
    print "local names:", locals()
```

After running

```
>> f(3)
local names: {'a': 2, 'x': 3}
```

NumPy / SciPy

See

`http://quant-econ.net/resources.html`

Go to NumPy / SciPy examples

Matplotlib Again

We've already created a few figures using Matplotlib

Matplotlib has a split personality

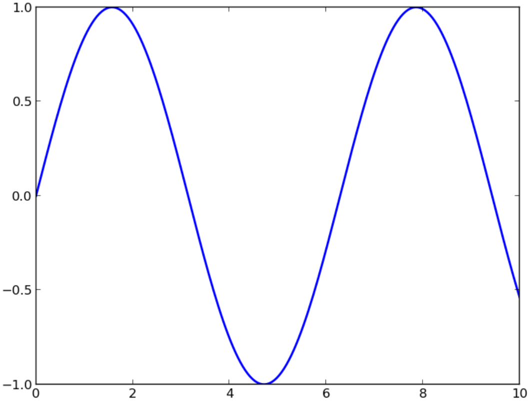
- API 1: Designed to mimic MATLAB graphics functionality
- API 2: Object oriented, and more Pythonic

MATLAB Style Plotting

The module **pylab** combines Matplotlib, NumPy and SciPy

- simple, straightforward, familiar to MATLAB users

```
from pylab import *
x = linspace(0, 10, 200)
y = sin(x)
plot(x, y, 'b-', linewidth=2)
show()
```


1.0
0.5
0.0
−0.5
−1.0
0
2
4
6
8
10

This coding style is easy but slightly un-Pythonic

```
from pylab import *
x = linspace(0, 10, 200)
y = sin(x)
plot(x, y, 'b-', linewidth=2)
show()
```

We're pulling lots of names into the global namespace

Lot of implicit calls behind the scenes

Matplotlib’s Object Oriented API

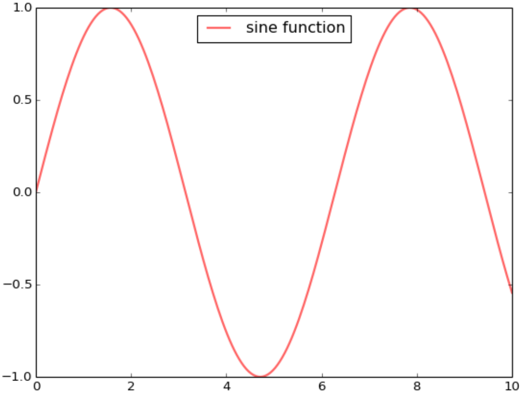
More typing, but cleaner, more control, more flexibility

An example, from plot_example_1.py:

```
import matplotlib.pyplot as plt
import numpy as np
fig, ax = plt.subplots()
x = np.linspace(0, 10, 200)
y = np.sin(x)
ax.plot(x, y, 'b-', linewidth=2)
plt.show()
```

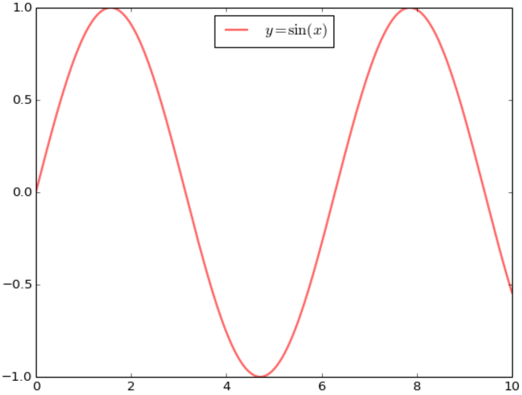
Red line, with legend (plot_example_2.py)

```python
import matplotlib.pyplot as plt
import numpy as np
fig, ax = plt.subplots()
x = np.linspace(0, 10, 200)
y = np.sin(x)
ax.plot(x, y, 'r-', lw=2, label='sine function', alpha=0.6)
ax.legend(loc='upper center')
plt.show()
```


sine function
1.0
0.5
0.0
−0.5
−1.0
0
2
4
6
8
10

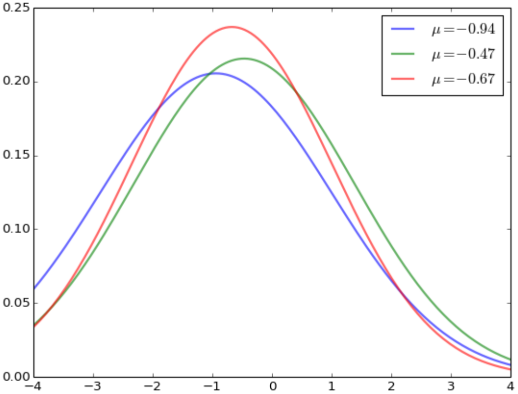
Using LaTeX (plot_example_3.py)

```
import matplotlib.pyplot as plt
import numpy as np
fig, ax = plt.subplots()
x = np.linspace(0, 10, 200)
y = np.sin(x)
ax.plot(x, y, 'r-', lw=2, label=r'$y=\sin(x)$', alpha=0.6)
ax.legend(loc='upper center')
plt.show()
```


$y = \sin(x)$
1.0
0.5
0.0
−0.5
−1.0
0
2
4
6
8
10

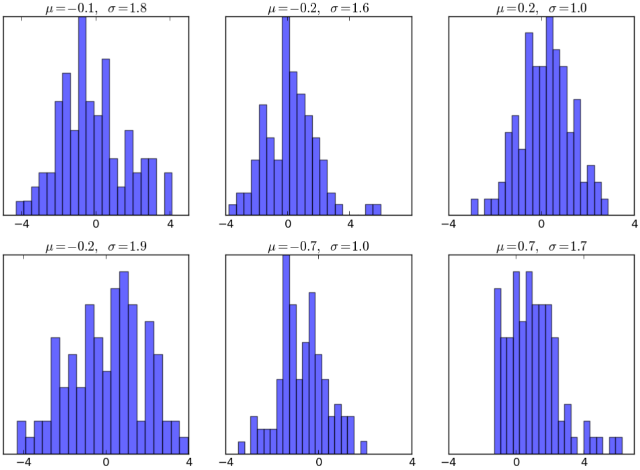
Multiple plots on one axis (plot_example_4.py)

```
import matplotlib.pyplot as plt
import numpy as np
from scipy.stats import norm
from random import uniform
fig, ax = plt.subplots()
x = np.linspace(-4, 4, 150)
for i in range(3):
    m, s = uniform(-1, 1), uniform(1, 2)
    y = norm.pdf(x, loc=m, scale=s)
    current_label = r’$\mu = {0:.2f}$’.format(m)
    ax.plot(x, y, lw=2, alpha=0.6, label=current_label)
ax.legend()
plt.show()
```


0.25
0.20
0.15
0.10
0.05
0.00
−4
−3
−2
−1
0
1
2
3
4
$\mu = -0.94$
$\mu = -0.47$
$\mu = -0.67$

Multiple axes on one figure (plot_example_5.py)

```
import matplotlib.pyplot as plt
import numpy as np
from scipy.stats import norm
from random import uniform
num_rows, num_cols = 2, 3
fig, axes = plt.subplots(num_rows, num_cols, figsize=(12, 8))
for i in range(num_rows):
    for j in range(num_cols):
        m, s = uniform(-1, 1), uniform(1, 2)
        x = norm.rvs(loc=m, scale=s, size=100)
        axes[i, j].hist(x, alpha=0.6, bins=20)
        t = r'$\mu = {0:.1f},\; \sigma = {1:.1f}$'.format(m, s)
        axes[i, j].set_title(t)
        axes[i, j].set_xticks([-4, 0, 4])
        axes[i, j].set_yticks([])
plt.show()
```


$\mu = -0.1, \quad \sigma = 1.8$
$\mu = -0.2, \quad \sigma = 1.6$
$\mu = 0.2, \quad \sigma = 1.0$
$\mu = -0.2, \quad \sigma = 1.9$
$\mu = -0.7, \quad \sigma = 1.0$
$\mu = 0.7, \quad \sigma = 1.7$
−4
0
4

Coding Style and PEP8

Python strongly favors consistency

Consistency is important in programming and mathematics

- Imagine a maths paper that swaps symbols $\cup$ and $\cap$

Standard Python style is set out in PEP8

http://www.python.org/dev/peps/pep-0008/

Please follow it!

More details:

- quant-econ.net/python_essentials.html
- quant-econ.net/python_oop.html
- quant-econ.net/python_foundations.html
- quant-econ.net/numpy.html
- quant-econ.net/scipy.html
- quant-econ.net/matplotlib.html

Exercise:

Ex. 1 of quant-econ.net/finite_markov.html