PyShop Session 2

Packages, packages, packages...

Tyler Abbot

Department of Economics
Sciences Po

Fall 2015

Outline

Why do we use packages?

- Packages are modular code
- They are general and can be applied easily
- We are not computer scientists!
- The number one rule of programming: "Never do anything twice."

What is NumPy?

- NumPy is the numpy ndarray.
- Optimized for vector operations
- Precompiled C allows you to write Python but run C
- The ndarray accepts more data types than the native Python implimentation
- Array indexing is more succinct than lists
- Plays very well with SciPy

Array Creation

```python
import numpy as np

x = np.array([1, 2, 3])
y = np.zeros((2, 2))
z = np.ones((2, 2))
a = np.eye((2))
```

- There are many ways to create arrays, so check them out in the documentation

Array Indexing

```
a = np.eye((2))
print a[0, :]
print a[1, :]

Output:
[ 1.  0.]
[ 0.  1.]
```

- Like lists, begins at 0
- Syntax slightly easier than lists
- Can slice arrays like lists

Pointers and Copies

```
a = np.eye(2)
b = a
print a
print b

Output:
[[ 1.  0.]
 [ 0.  1.]]
[[ 1.  0.]
 [ 0.  1.]]

a[0, 0] = 0
print a
print b
```

Pointers and Copies II

```
Output:
[[ 0.  0.]
 [ 0.  1.]]
[[ 0.  0.]
 [ 0.  1.]]

a = np.eye(2)
b[:] = a
c = a.copy()
print a
print b
print c

Output:
[[ 1.  0.]
 [ 0.  1.]]
[[ 1.  0.]
 [ 0.  1.]]
[[ 1.  0.]
 [ 0.  1.]]
```

Pointers and Copies III

- Python avoids copies at all costs!
- If you want to create a new variable identical to an old one, you need to explicitly tell Python
- Use the `.copy()` method that most objects have, take a slice over the entire NEW object, or use the `copy()` function

Broadcasting

- Telling NumPy to work on arrays of different dimensions doesn't typically generate an error
- For instance, scalar multiplication
- "Broadcasting" - how NumPy deals with this issue
- Next week: deeper discussion of this issue

```
a = np.ones((2, 2))
b = 3

print a + b

Output:

[[ 4.  4.]
 [ 4.  4.]]
```

Speed

- The beauty of NumPy is speed
- Python generally slower than C, Fortran, etc.
- NumPy takes a step to solve this problem
- Very smart people spent a lot of time thinking about this...

```
Speed test if you have Ipython Notebook open
```

Is that it!?

No.

- NumPy is mainly the ndarray
- Necessary to talk about SciPy to do anything more than define arrays
- Next week we'll talk about linear algebra, sorting, re-sizing, and random variables in NumPy

The SciPy Library

- A bunch of numerical algorithms
- Used by many packages, so whatever you do you'll need it
- Keys for you:
  `integrate`, `interpolate`, `optimize`, `sparse`, `stats` subpackages
- Today: integration, unconstrained minimization, root finding, interpolation

Integration

- `scipy.integrate` is the main integration package
- Offers basic quadrature, gaussian quadrature, simpson's rule, trapezoid rule, Rhomberg integration, all in high dimensions
- For general integration, offers `quad`, `dblquad`, `tplquad`, `nquad` for one, two, three, and n dimensional integration

Integration II

An Example

The pdf of an exponentially distributed rdv with parameter k is

$$f(x,k) = ke^{-kx}$$
$$x \in [0, \infty)$$

```
import scipy.integrate

def integrand(x, k):
    return k*np.exp(-k*x)

k = 1.0
scipy.integrate.quad(integrand, 0, np.inf, args = (k))

Output:
(1.0000000000000002, 5.842606742906004e-11)
```

Integration III
An Example

Alternatively, in one line:

```
scipy.integrate.quad(lambda x: k*np.exp(-k*x), 0, np.inf)

Output:
(1.0000000000000002, 5.842606742906004e-11)
```

- Lambda funcitons are a way to define functions inline
- Useful for simple operations
- Keeps the namespace clear

Optimization

- SciPy offers many methods for optimization and root finding
  - Local optimization
  - Equation minimizers
  - Global opimization
  - Curve fitting (non-linear least squares)
  - Scalar and multidimensional root finding
  - Linear programming
- The optimize subpackage also offers several useful utilities for your own opimization algorithms: gradient estimation by finite differencing, line search, l-bfgs estimation of the hessian (for very high dimensional problems), etc.

- Only available for scalar functions (in fact, doesn't really make sense for multiple dimensions...)
- Fun fact: the Rosenbrock function is a non-convex function used as a performance test for optimization.
- Also known as Rosenbrock's banana function

$$f(x, y) = (a - x)^2 + b(y - x^2)^2$$

- The solution is known as $(x, y) = (a, a^2)$.

Optimization III

Unconstrained Minimization (cont'd)

```
import scipy.optimize
import time

def rosenbrock(x, a, b):
    return (a - x[0])**2 + b*(x[1] - x[0]**2)**2

a = 1.
b = 100.

x0 = np.array([2., 3.])

t0 = time.time()
res = scipy.optimize.minimize(rosenbrock, x0, args=(a, b),
                              method='Nelder-Mead')
t1 = time.time()
print "\nProcess executed in : %s : seconds.\n" %(t1 - t0)
print res
```

Optimization IV

Unconstrained Minimization (cont'd)

```
Output:

Process executed in : 0.00798296928406 : seconds.

  status: 0
    nfev: 158
 success: True
     fun: 2.1717765323851955e-10
       x: array([ 0.99998529,  0.99997065])
 message: 'Optimization terminated successfully.'
     nit: 84
```

Root Finding I

- In general, the syntax similar for scalar and vector valued functions
- Can use `root` funciton for both scalars and vectors
- Offers many different methods for the jacobian: hybrid, broyden, anderson, krylov, etc.

Let's try to find the root of

$$f(x,y) = \begin{bmatrix} f_1(x,y) \\ f_2(x,y) \end{bmatrix} = \begin{bmatrix} a(1-x) \\ b(y-x^2)^2 \end{bmatrix}$$

Where a and b are constants. Notice that these two functions are the terms of the Rosenbrock funciton. Whill the solution be the same!? Let's find out.

Root Finding III

An Example (cont'd)

```
def f(x, a, b):
    return np.array([a*(1 - x[0]), b*(x[1] - x[0]**2)**2])

a = 1.
b = 100.
x0 = np.array([10., 2.])

sol = scipy.optimize.root(f, x0, args=(a, b), method='hybr')
print sol
```

As with minimization, there are several methods available.

Root Finding IV

An Example (cont'd)

```
Output:
  status: 1
 success: True
     qtf: array([  4.43734258e-29,  -2.81227533e-33])
    nfev: 90
       r: array([  1.57784785e+04,  -2.68667616e-13,
                   1.00571850e-17])
     fun: array([  0.00000000e+00,   1.10933565e-29])
       x: array([ 1.,  1.])
 message: 'The solution converged.'
    fjac: array([[ -6.33774668e-05,   9.99999998e-01],
       [ -9.99999998e-01,  -6.33774668e-05]])
```

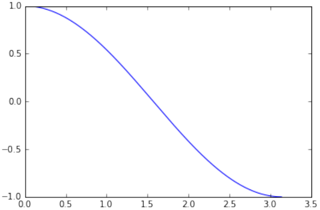
SciPy Conclusion

- There are many possibilities
- Understanding SciPy is important for understanding all other numerical packages
- Next week we'll solve a more complex problem using SciPy and NumPy (per Nicolo's request: maximum likelihood Probit!)

- Originally meant to match plotting capabilities of MATLAB
- More object oriented
- Fully customizable
- Easy to use

Simply use the `pylab` function.

```
import matplotlib.pyplot as plt
%matplotlib inline

x = np.arange(0, np.pi, 0.01)
y = np.cos(x)
plt.plot(x, y)
plt.show()
```



Customizing Plot Attributes I

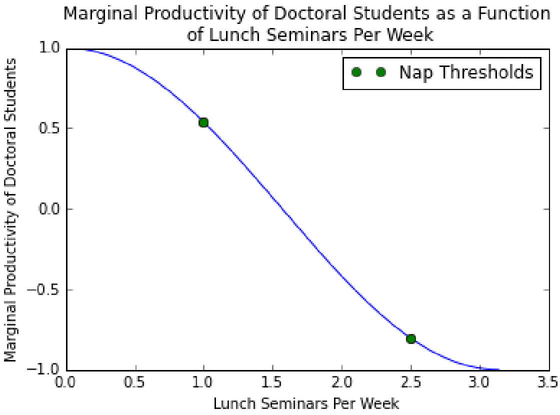
- The plot you've created is an object, whose attributes you can modify
- Many things to customize, but the syntax is always the same: `plt.attribut(value)`

Let's customize our graph so that it represents something we understand:

Customizing Plot Attributes II

```
#Add axis labels
plt.xlabel('Lunch Seminars Per Week')
plt.ylabel('Marginal Productivity of Doctoral Students')

#Add title
plt.title("Marginal Productivity of Doctoral Students as a Function\n"
          + " of Lunch Seminars Per Week")

#Add emphasis to important points
points = np.array([1.0, 2.5])
plt.plot(points, np.cos(points), 'ro')

#Add a label and legend to the points
plt.plot(points, np.cos(points), 'o', label='Nap Thresholds')
plt.legend()

#But the legend is poorly placed, so move it to a better spot
plt.legend(loc=0)

plt.show()
```

Customizing Plot Attributes III



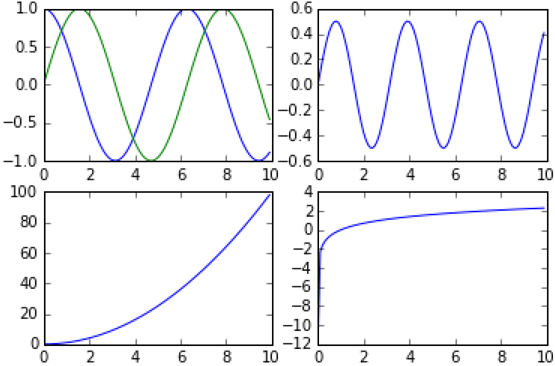
Subplotting

- Under the hood, pyplot is object oriented
- Every plot object is a figure object and some number of axes objects
- The figure contains information about layout of the axes and axes contain the individual plotting information
- I encourage you to define these yourself, so you have a better understanding of the mechanics
- Let's see a subplots example

Customizing Plot Attributes II

```
x = np.arange(0, 10, 0.1)
f = lambda x: np.cos(x)
g = lambda x: np.sin(x)
h = lambda x: x**2
i = lambda x: np.log(x + 0.00001)

#Create the figure and axes objects.
fig, ((ax1, ax2), (ax3, ax4)) = plt.subplots(2, 2)

#Plot on the axes objects.
ax1.plot(x, f(x))
ax1.plot(x, g(x))
ax2.plot(x, f(x)*g(x))
ax3.plot(x, h(x))
ax4.plot(x, i(x))

plt.show()
```

Customizing Plot Attributes III



3-Plotting

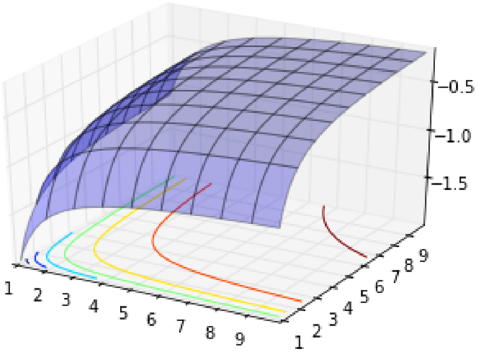
- Easy to use
- Powerful for visualizing a function
- Utility functions and indifference curves become very clear!

$$U(c_1, c_2) = \frac{c_1^{1-\gamma}}{1-\gamma} + \beta \frac{c_2^{1-\gamma}}{1-\gamma}$$

Customizing Plot Attributes II

```
from mpl_toolkits.mplot3d import Axes3D
def U(c1, c2, beta, gamma):
    return c1**(1 - gamma)/(1 - gamma) + beta*c2**(1 - gamma)/(1 - gamm

beta, gamma = 0.98, 2.0
low, high = 1.0, 10.0

fig = plt.figure()
ax = fig.gca(projection="3d")

c1, c2 = np.arange(low, high, 0.1), np.arange(low, high, 0.1)
C1, C2 = np.meshgrid(c1, c2)

utils = U(C1, C2, beta, gamma)

ax.plot_surface(C1, C2, utils, alpha=0.3)
cset = ax.contour(C1, C2, utils, zdir='z', offset=-2.0)

plt.show()
```

Customizing Plot Attributes III



The DataFrame

- Key to Pandas success
- Allows columns to vary in type
- Can use heirarchical indexing
- Built in methods allow for quick data management

Data IO

- Can read/write to several data types
- Easily reference url or file name
- Works with HDF5, a high performance computing datatype for large data sets

DataFrames

Go to Ipython Notebook

Conclusion

- NumPy is very fast and useful for vectorized and numerical calculation
- SciPy is gigantic!
- matplotlib can be quite easy to use, but is very customizable
- Pandas is often clunky, but with practice you might like it?