Python for Scientific Computing

Erik Spence

SciNet HPC Consortium

13 July 2016

Material for this class

All the material for the HPC Summer School can be found here:

https://wiki.scinet.utoronto.ca/wiki/index.php/2015_Ontario_Summer_School_for_High_Performance_Computing_Central

The slides for this class can be found here:

http://tinyurl.com/ss2016-P1

and at the SciNet education website:

http://support.scinet.utoronto.ca/education

Today's class

Today we will discuss the following topics:

- Getting Python started.
- Primitive types.
- Lists.
- Arrays.
- Plotting.
- Statistics.
- Data frames.
- Functions.

In this class we will assume some basic understanding of Python, but not much. It will focus on those aspects of Python which are relevant to Scientific programming. Feel free to ask if you don't understand something.

Notes about Python

Things to know about Python:

- First released in February 1991.
- Python combines functional and syntactic aspects of the languages ABC, C, Modula-3, SETL, Lisp, Haskell, Icon, Perl.
- Python is a high-level, interpreted language.
- Python supports many programming paradigms (procedural, object-oriented, functional, imperative).
- Python variables are dynamic, meaning they merely labels for a typed value in memory. They are easily re-assigned to refer to some other memory location.
- Python has automatic memory management, and garbage collection.
- Python is case sensitive.
- Python 3.X is not back-compatible with Python 2.X.

Notes about Python, continued

Some important things to know about Python:

- Python is a scripting language, meaning an interpreter executes commands one line at a time (not a compiled language).
- Python can be used interactively, with or without IDE (IDLE).
- Python has a large repository of community packages.
- Python is a general purpose language; it was not designed with data analysis in mind.
  - Several important features, such as numerics, are add-ons.
  - Visualization can be annoyingly tricky.
  - But can also be useful outside of number crunching.
- Because Python is a general-purpose language it tends to be more versatile and flexible than R.

Starting Python

How you start Python interpreter depends upon your system:

- Windows: several graphical Python interfaces exist (IDLE, Anaconda). Launch whichever one you have installed.
- Mac: similar to Windows, but running from the command line is also an option.
- Linux: open a terminal. Type "python" (or "ipython"). Use "exit" to quit.

Open up your Python interface now. Raise your hand if you don't think it's working. Please follow along by entering the commands on the slides, and playing with the output.

Starting Python on SciNet

Alternatively, you can log into SciNet and run Python there.

```
ejspence@mycomp ~>
ejspence@mycomp ~> ssh ejspence@login.scinet.utoronto.ca -X
ejspence@scinet01-ib0 ~>
ejspence@scinet01-ib0 ~> ssh -X gpc03
ejspence@gpc-f103n084-ib0 ~>
ejspence@gpc-f103n084-ib0 ~> module load intel/15.0.2 python/2.7.8
ejspence@gpc-f103n084-ib0 ~>
ejspence@gpc-f103n084-ib0 ~> ipython --pylab
In [1]:
```

We won't be using parallel Python capabilities until this afternoon, so you should be able to just work either on your own laptop, or on a SciNet devel node for this morning.

Python versus ipython

There are two main ways to run Python interactively: regular Python and IPython ("Interactive Python"). They both have advantages and disadvantages:

- Regular Python is what you get when you type "python" at the command prompt.
- There aren't as many nice features built into regular Python.
- But regular Python is what you get when you run Python scripts, so you're sure to get the right answer the first time, since there aren't nice features included which are available in regular Python.
- IPython has tab line completion built in, interactive plotting (when "–pylab" is invoked at the command line) and other features.
- But IPython is not what you have when you run scripts.

Python data types

Python has several standard data types:

- Numbers
- Strings
- Booleans
- Container types
    - Lists
    - Sets
    - Tuples
    - Dictionaries

We aren't going to cover all of the data and container types, since we're focusing on Scientific data analysis.

Integers in Python

Python offers two default types of integers:

- "plain integers":
  - All integers are plain by default unless they are too big.
  - These are implemented using long integers in C. This gives them, depending on the system, at least 32 bits of range.
  - The maximum value can be found by checking the sys.maxint value.

```
In [1]: import sys
In [2]: print sys.maxint
2147483647
In [3]: a = 10
In [4]: type(a)
Out[4]: int
In [5]: int(10.0)
Out[5]: 10
```

Importing modules

Python offers several ways to import packages:

- "import numpy"
- "import numpy as np"
- "from numpy import linalg"
- "from numpy import linalg, fft, random"
- "from numpy import linalg as la"

```
In [6]:
In [6]: import os
In [7]: print os.name
posix
In [8]: import time as t
In [9]: t.time()
Out[9]: 1436274243.333076
In [10]: from calendar import isleap
In [11]: isleap(2114)
Out[11]: False
In [12]:
```

Installing packages

Installing packages is not quite a easy as in R. The package manager usually used is "pip". To install, say, the numpy package, in Linux and on a Mac, open a terminal, and use the command

```
ejspence@mycomp ~> pip install numpy
```

or if you don't have administrator permission

```
ejspence@mycomp ~> pip install --user numpy
```

On Windows, search for the "cmd" prompt in the Start menu. Then type these commands:

```
C:\Users\ejspence>
C:\Users\ejspence> cd C:\Python27\Scripts
C:\Python27\Scripts>
C:\Python27\Scripts> pip.exe install numpy
```

Getting help

Like R, once a module or package has been imported, you can get information about how to run that package:

- the "help" command gives information about the package or function.
- the "?" command gives installation information and the package docstring.

Type 'q' to get out of either of these, at least on Linux machines.

```
In [12]:
In [12]: import os
In [13]: help(os)

:

In [14]:
In [14]: ? time

:

In [15]:
```

Integers in Python, continued

The second type of Python integer:

- "long integers":
  - Have infinite range.
  - Are invoked using the long(something) function, or by placing an "L" after the number.

```
In [15]: a = 10
In [16]: b = 10L
In [17]: b
Out[17]: 10L
In [18]: type(b)
Out[18]: long
In [19]: c = long(a)
In [20]: type(c)
Out[20]: long
```

Floats in Python

Python offers two types of floating point numbers:

- "floating point numbers":
  - Based on the C double type.
  - You can specify the exponent by putting "e" in your number.
  - Information about floats on your system can be found in sys.float_info.

```
In [21]: import sys
In [22]: print sys.float_info
sys.float_info(max=1.7976931348623157e+308, max_exp=1024, max_10_exp=308,
min=2.2250738585072014e-308, min_exp=-1021, min_10_exp=-307, dig=15,
mant_dig=53, epsilon=2.220446049250313e-16, radix=2, rounds=1)
In [23]:
In [23]: a = 4.5e245
In [24]: a
Out[24]: 4.5e+245
```

Floats in Python, continued

The second type of Python floating point number:

- "complex numbers":
  - Have a real and imaginary part, both of which are floats.
  - Use z.real and z.imag to access individual parts.

```
In [25]: a = complex(1.,3.0)

In [26]: print a
(1+3j)

In [27]:

In [27]: b = 1.0 + 2.j

In [28]: print b.imag
2.0

In [29]: complex(10.0)
Out[29]: (10+0j)
```

Booleans

Python supports standard boolean variables and operations.

```
In [30]: bool(1)
Out[30]: True

In [31]: bool(0)
Out[31]: False

In [32]: bool(3)
Out[32]: True

In [33]: True + 1
Out[33]: 2

In [34]: False + 1
Out[34]: 1

In [35]:
```

```
In [35]: a = True

In [36]: a and False
Out[36]: False

In [37]: not a
Out[37]: False

In [38]: a or False
Out[38]: True

In [39]: a & True
Out[39]: True

In [40]: a | True
Out[40]: True
```

Strictly speaking, booleans are a sub-type of plain integers. Note that the values are "True", rather than "TRUE" as in R. Note also that you must type the full word "True"; "T" is insufficient.

Boolean bitwise operations

Python contains the usual bitwise operations.

Operation	Equivalent	Result
x | y	x or y	bitwise OR of x and y
x ^ y		bitwise XOR of x and y
x & y	x and y	bitwise AND of x and y
x << n		x shifted left by n bits
x >> n		x shifted right by n bits
~x		x bitwise inverted

And the usual testing operators.

Operation	Equivalent	Result
!x	not x	the opposite of x
x == y		is x equal to y?
x != y		is x not equal to y?

Strings

Strings are delimited by single or double quotation marks (' or "):

```
In [41]: word = "Hello World"
In [41]: word
Out[42]: "word"
In [43]: print word
Hello World
In [44]: print word + " again!"
Hello World again!
In [45]:
```

Python has a tonne of string manipulation features built into it. If your work involves string manipulation, Python isn't a bad choice.

Lists

Lists are similar to lists in R.

```
In [45]: a = [3.123, "Hello World", 3]

In [46]: print a
[3.123, 'Hello World', 3]

In [47]:

In [47]: L = ['yellow', 'red', 'blue', 'green', 'black']

In [48]: print len(L)
5

In [49]: L.append('orange')

In [50]: print L
['yellow', 'red', 'blue', 'green', 'black', 'orange']

In [51]:
```

Python has a tonne of list manipulation features built into it. If you need to manipulate a list in some way, look up the options, there are many.

Looping over lists

A very common operation is to loop over lists:

```
In [51]:
In [51]: a
Out[51]: [1, 2, 3, 4]
In [52]:
In [52]: for item in a:
...:         print item
...:
1
2
3
4
In [52]:
```

Note that weird whitespace before the 'print' statement.

Whitespace

All code blocks in Python are delineated by white space. But the amount of indentation does not need to be the same from one code block to another. But it must be consistent within the same code block.

```
In [52]: for i in range(5):
...:               if (i > 2):
...:                      print 'eek'
...:               elif (i == 2):
...:                                     print i
...:               else:
...:                print i - 7
...:
-7
-6
2
eek
eek
```

```
In [53]:
```

Loops

Like all languages, Python has the usual loops.

```
In [53]:
In [53]: for i in range(5,10,2):
...:    print i
...:
5
7
9
In [54]:
In [54]: i = 0
In [55]: while (i < 2):
...:        print i
...:        i += 1
...:
0
1
In [56]:
```

List comprehensions

Python allows you to create lists with loops, in what is at first a somewhat strange syntax:

```
In [56]:
In [56]: S = [x**2 for x in range(10)]
In [57]:
In [57]: S
Out[58]: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
In [59]:
In [59]: t = [x**2 for x in range(10) if x > 5]
In [60]: t
Out[61]: [36, 49, 64, 81]
In [62]:
```

These are called "list comprehensions". The basic syntax is

[expression(item) for item in list conditional(item)].

Dictionaries

Dictionaries are a Python data type which associates keys to values. These definitions of the dictionary are all equivalent:

```
In [62]:
In [62]: a = dict(one = 1, two = 2, three = 3)
In [63]: b = 'one':  1, 'two':  2, 'three':  3
In [63]: e =
In [64]: e['one'] = 1
In [65]:
In [65]: a
'one': 1, 'three': 3, 'two': 2
In [66]:
In [66]: a['three']
3
In [67]:
```

Lists aren't the ideal data type

Lists can do funny things that you don't expect, if you're not careful.

- Lists are just a collection of items, of any type.
- If you do mathematical operations on a list, you won't get what you expect.
- These are not the ideal data type for scientific computing.
- Arrays are a much better choice, but are not a native Python data type.

```
In [67]:
In [67]: a = [1, 2, 3]
In [68]: a
Out[68]: [1, 2, 3]
In [69]: a.append(4)
In [70]: a
Out[70]: [1, 2, 3, 4]
In [71]:
In [71]: b = [3, 5, 5, 6]
In [72]: b
Out[72]: [3, 5, 5, 6]
In [73]: 2 * a
Out[73]: [1, 2, 3, 4, 1, 2, 3, 4]
In [74]: a + b
Out[74]: [1, 2, 3, 4, 3, 5, 5, 6]
In [75]:
```

Arrays are what we want to use

Almost everything that you want to do starts with NumPy.

- Contains arrays of various types and forms: zeros, ones, linspace, *etc.*
- linspace takes 2 or 3 arguments, the default number of entries is 50.

```
In [75]: from numpy import zeros,
  ones, linspace, array
```

```
In [76]: zeros(4)
Out[76]: array([ 0., 0., 0., 0.])
```

```
In [77]: ones(5, dtype = int)
Out[77]: array([ 1, 1, 1, 1, 1])
```

```
In [78]: zeros([2,2])
Out[78]:
 array([[ 0.,  0.],
        [ 0.,  0.]])
```

```
In [79]: arange(5)
Out[79]: array([ 0, 1, 2, 3, 4])
```

```
In [80]: linspace(1, 5)
Out[80]: array([ 1., 1.08163265,
1.16326531, 1.24489796,
.
.
4.67346939, 4.75510204,
4.83673469, 4.91836735, 5.  ])
```

```
In [81]: linspace(1, 5, 6)
Out[81]: array([ 1., 1.8, 2.6, 3.4,
  4.2, 5.])
```

```
In [82]:
```

Accessing array elements

Elements of arrays are accessed using square brackets.

- Like most languages, the first index is the row, the second is the column.
- Indexing starts at 0.

```
In [82]: zeros([2, 3])
Out[82]:
 array([[ 0.,  0.,  0.],
        [ 0.,  0.,  0.]])
```

```
In [83]: a = zeros([2, 3])
```

```
In [84]: a[1,2] = 1
```

```
In [85]: a[0,1] = 2
```

```
In [86]:  a
Out[86]:
 array([[ 0.,  2.,  0.],
        [ 0.,  0.,  1.]])
```

```
In [87]: a[2,1] = 1
------------------------------------
IndexError Traceback
<ipython-input-21-83f146d6c508> in
<module>()
----> 1 a[2,1] = 1
IndexError:  index (2) out of range
(0<=index<2) in dimension 0
```

```
In [88]:
```

Indexing arrays

Notes about Python indexing:

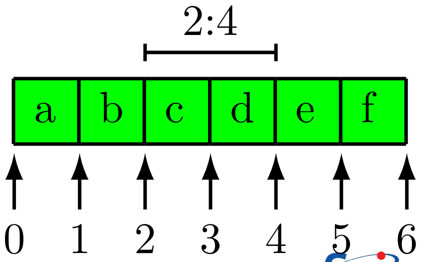
- Read "2:4" as "from the beginning of the second element, to the beginning of the fourth element".
- Negative indexing is supported, with very different behaviour than R.
- If a third index is specified, it refers to the step size ("1:10:2", for example).
- If no index is specifed, either "beginning" or "end" is assumed.

```
In [88]: a = array([1,2,3,4,5,6,7])
```

```
In [89]: len(a)
Out[89]: 7
```

```
In [90]: print a[6]
7
```

```
In [91]: print a[6:7]
[7]
```



Indexing arrays, continued

There are many ways to select and slice elements from an array:

```
In [92]: a = array([1,2,3,4,5,6,7])
```

```
In [93]: print a[:4]
[1 2 3 4]
```

```
In [94]: print a[4:]
[5 6 7]
```

```
In [95]: print a[-3:]
[5 6 7]
```

```
In [96]: print a[::2]
[1 3 5 7]
```

```
In [97]: a[-1]
Out[97]: 7
```

What is the output of a[-5:-4] and a[-4:-5]?

Indexing arrays, continued more

Elements in an array can be selected using a boolean array. Boolean arrays can be created using a conditional expression.

```
In [98]:
In [98]: a = arange(5)
In [99]: a
Out[99]: array([0, 1, 2, 3, 4])
In [100]: a > 2
Out[100]: array([False, False, False, True, True], dtype=bool)
In [101]: a[a > 2]
Out[101]: array([3, 4])
In [102]: a[(a % 2) == 0]
Out[102]: array([0, 2, 4])
In [103]:
```

The "%" symbol is the modulus operator.

Pop quiz!

Use the "numpy.random.random_integers" function to create vector of random integers, of length 16, with elements in the range 0 to 30. If an element is divisible by 3, set it to -1.

Pop quiz!

Use the "numpy.random.random_integers" function to create vector of random integers, of length 16, with elements in the range 0 to 30. If an element is divisible by 3, set it to -1.

```
In [107]:
In [107]: import numpy as np
In [108]:
In [108]: a = np.random.random_integers(0, 30, 16)
In [109]:
In [109]: a[a % 3 == 0] = -1
In [110]:
In [110]: a
Out[110]: array([19, 14, 1, 23, 2, 7, 8, -1, -1, -1, 11, 8, 1, 28, -1, 1])
In [111]:
```

vector-vector & vector-scalar multiplication

1D arrays are often called 'vectors'.

- When vectors are multiplied using "*" you get element-by-element multiplication.
- When vectors are multiplied by a scalar (a 0-D array), you also get element-by-element multiplication.

```
In [111]: a = arange(4)
In [112]: a
Out[112]: array([0, 1, 2, 3])
In [113]: b = arange(4.) + 3
In [114]: b
Out[114]: array([ 3., 4., 5., 6.])
In [115]: c = 2
In [116]: c
Out[116]: 2
In [117]: a * b
Out[117]: array([ 0., 4., 10., 18.])
In [118]: a * c
Out[118]: array([0, 2, 4, 6])
In [119]: b * c
Out[119]: array([ 6., 8., 10., 12.])
In [120]:
```

vector-vector outer product

Outer products of two vectors are also possible.

```
In [120]: from numpy import outer
In [121]:
In [121]: a = array([1,2,4])
In [122]: b = array([1,3,2])
In [123]:
In [123]: outer(a,b)
Out[123]:
 array([[ 1,  3,  2],
        [ 2,  6,  4],
        [ 4, 12,  8]])
In [124]:
```

matrix-vector multiplication

A 2D array is sometimes called a 'matrix'.

- Matrix-scalar multiplication is element-by-element.
- Matrix-vector multiplication DOES NOT give the standard result!

```
In [124]: a = array([[1,2,3], [2,3,4]])
In [125]: a
Out[125]:
 array([[1, 2, 3],
        [2, 3, 4]])
In [126]: b = arange(3) + 1
In [127]: b
Out[127]: array([1, 2, 3])
In [128]: a * b
Out[128]:
 array([[1, 4, 9],
        [2, 6, 12]])
```

Normal matrix-vector multiplication:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} * \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} = \begin{bmatrix} a_{11} * b_1 + a_{12} * b_2 + a_{13} * b_3 \\ a_{21} * b_1 + a_{22} * b_2 + a_{23} * b_3 \\ a_{31} * b_1 + a_{32} * b_2 + a_{33} * b_3 \end{bmatrix}$$

matrix-matrix multiplication

Not surprisingly, matrix-matrix multiplication doesn't work as expected either, instead doing the same thing as vector-vector multiplication.

```
In [129]: a = array([[1,2,3], [2,3,4]])
```

```
In [130]: b = array([[1,2,3], [2,3,4]])
```

```
In [131]: a
Out[131]:
 array([[1, 2, 3],
        [2, 3, 4]])
```

```
In [132]: a * b
Out[132]:
 array([[1, 4, 9],
        [4, 9, 16]])
```

Normal matrix-matrix multiplication:

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} * \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} =$$

$$\begin{bmatrix} a_{11} * b_{11} + a_{12} * b_{21} & a_{11} * b_{12} + a_{12} * b_{22} \\ a_{21} * b_{11} + a_{22} * b_{21} & a_{21} * b_{12} + a_{22} * b_{22} \end{bmatrix}$$

How to do matrix algebra?

There are two solutions to these matrix multiplication problems.

- The specially built-in array fixes (using 'array' types).
- The matrix module (using 'matrix' types).

The latter option is a bit clunkier, so we recommend the 'fixes'.

```
In [133]: from scipy import dot
In [134]: a = array([[1,2,3],[2,3,4]])
In [135]: b = array([[1,2,3],[2,3,4]])
In [136]: a
Out[136]:
 array([[1, 2, 3],
        [2, 3, 4]])
```

```
In [137]: a.transpose()
Out[137]:
 array([[1, 2],
        [2, 3],
        [3, 4]])
In [138]: dot(a.transpose(), b)
Out[138]:
 array([[5,  8, 11],
        [8, 13, 18],
       [11, 18, 25]])
In [139]: dot(b, a.transpose())
Out[139]:
 array([[14, 20],
        [20, 29]])
In [140]: c = arange(3) + 1
In [141]: dot(a,c)
Out[141]: array([14, 20])
```

The linalg module

The linalg module contains useful functions for matrix algebra.

- Typical matrix functions: inv, det, norm...
- More advanced functions: eig, SVD, cholesky...
- Both NumPy and SciPy have a linalg module. Use SciPy, because it is always compiled with BLAS/LAPACK support.

```
In [142]: from scipy import dot, linalg

In [143]: a = array([[1,2,3], [3,4,5], [1,1,2]])

In [144]: linalg.det(a)
Out[144]: -2.0

In [145]: dot(a, linalg.inv(a))
Out[146]:
 array([[ 1.00000000e+00,  0.00000000e+00,  0.00000000e+00],
        [2.77555756e-16,  1.00000000e+00,  0.00000000e+00],
        [0.00000000e+00,  5.55111512e-17,  1.00000000e+00]])
```

Solving systems of equations

The linalg module comes with an important function: solve.

- linalg.solve is used to solve the system of equations $Ax = b$.

```
In [147]: from scipy import linalg
In [148]: a = array([[1,2,3],
  [3,4,5], [1,1,2]])
In [149]: a
Out[149]:
 array([[1, 2, 3],
        [3, 4, 5],
        [1, 1, 2]])
In [150]:
```

```
In [150]: b = array([3, 4, 2])
In [151]: b
Out[151]: array([3, 4, 2])
In [152]:
In [152]: x = linalg.solve(a, b)
In [153]: x
Out[153]: array([-0.5, -0.5, 1.5])
In [154]:
```

$$\begin{bmatrix} 1 & 2 & 3 \\ 3 & 4 & 5 \\ 1 & 1 & 2 \end{bmatrix} * \begin{bmatrix} -0.5 \\ -0.5 \\ 1.5 \end{bmatrix} = \begin{bmatrix} 3 \\ 4 \\ 2 \end{bmatrix}$$

Functions

Functions are created with the "def" command.

```
In [154]: def squared(x):
...:            return x**2
...:
In [155]:
In [155]: squared(3)
Out[156]: 9
In [157]:
In [157]: def hello(a, b = "pants"):
...:            print a + b
...:
In [158]:
In [158]: hello('I')
Ipants
In [159]: hello('I', b = 'robot')
Irobot
In [160]:
```

Functions

Awesome functions should be tested and saved in a library of awesomeness. This is how you access it.

```
In [160]: import awesomecode

In [161]:

In [161]: awesomecode.squared(4)
16

In [162]:

In [162]: from awesomecode import hello

In [163]:

In [163]: hello('A')
Apants

In [164]:
```

```
# awesomecode.py

def squared(x):
  return x**2

def hello(a, b = "pants"):
  print a + b
```

Maplotlib's pyplot

The most commonly-used Python visualization package is matplotlib.pyplot:

- matplotlib is an add-on package to Python.
- Designed to have look which mimics MATLAB.
- Has all the plotting types you'd expect: line, scatter, bar, pie, contour, polar, box, *etc.*
- More-advanced functionality includes subplots, inset plots, colourbars, legends, *etc.*
- Control over every aspect of your plot is available, though not necessarily easily or obviously.
- Also has 3D plotting, built-in widgets, animations.

To learn more about visualization, attend the Visualization Summer School session on Friday.

Plotting in Python

By default, plotting in Python is a little wonky. Note that the following may not work in some IDEs.

```
ejspence@mycomp ~>
ejspence@mycomp ~> python
>>>
>>> import matplotlib.pyplot as plt
>>> plt.figure()
<matplotlib.figure.Figure object at 0x7fdbfec5ff90>
>>>
>>> plt.show()
>>>
```

As you can see, the default behaviour of matplotlib is to delay drawing the figure until it is told to do so. (Close the figure to get your prompt back.)

Plotting in Python, continued

Using ipython instead of python will get you around this, but only if you use the --pylab flag.

```
ejspence@mycomp ~>
ejspence@mycomp ~> ipython
In [1]:
In [2]: import matplotlib.pyplot as plt
In [3]: plt.figure()
<matplotlib.figure.Figure object at 0x7fdbfec5ff90>
In [4]: exit
ejspence@mycomp ~>
ejspence@mycomp ~> ipython --pylab
In [1]: import matplotlib.pyplot as plt
In [2]: plt.figure()
```

Interactive mode will only work with pyplot commands, however, not matplotlib objects.

Plotting

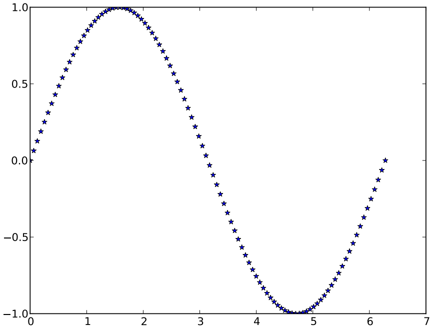
```
In [3]: import numpy as np

In [4]: import matplotlib.pyplot
  as plt

In [5]:
```

Plotting

```
In [3]: import numpy as np
In [4]: import matplotlib.pyplot
  as plt
In [5]: x =
  np.linspace(0, 2 * np.pi, 100)
In [6]:
```

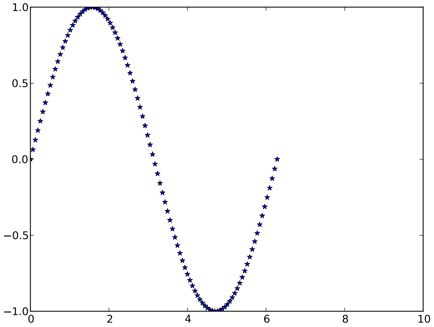
Plotting

```
In [3]: import numpy as np
In [4]: import matplotlib.pyplot
  as plt
In [5]: x =
  np.linspace(0, 2 * np.pi, 100)
In [6]: plt.plot(x, np.sin(x), '*')
In [7]:
```



Plotting

```
In [3]: import numpy as np
In [4]: import matplotlib.pyplot
  as plt
In [5]: x =
  np.linspace(0, 2 * np.pi, 100)
In [6]: plt.plot(x, np.sin(x), '*')
In [7]: plt.xlim(0, 10)
Out[7]: (0, 10)
In [8]:
```



Plotting

```
In [3]: import numpy as np
```

```
In [4]: import matplotlib.pyplot
  as plt
```

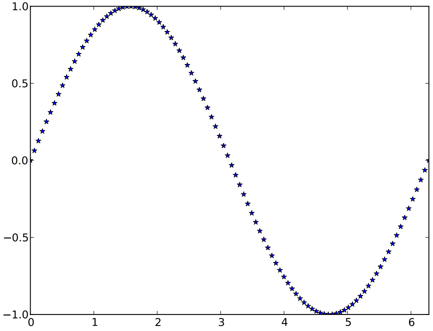
```
In [5]: x =
  np.linspace(0, 2 * np.pi, 100)
```

```
In [6]: plt.plot(x, np.sin(x), '*')
```

```
In [7]: plt.xlim(0, 10)
Out[7]: (0, 10)
```

```
In [8]: plt.xlim(0, 2 * np.pi)
Out[8]: (0, 6.2831853071795862)
```

```
In [9]:
```



Plotting

```
In [3]: import numpy as np
```

```
In [4]: import matplotlib.pyplot
  as plt
```

```
In [5]: x =
  np.linspace(0, 2 * np.pi, 100)
```

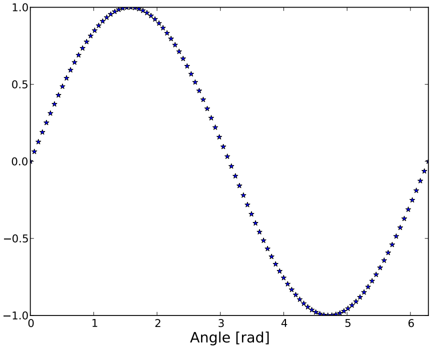
```
In [6]: plt.plot(x, np.sin(x), '*')
```

```
In [7]: plt.xlim(0, 10)
Out[7]: (0, 10)
```

```
In [8]: plt.xlim(0, 2 * np.pi)
Out[8]: (0, 6.2831853071795862)
```

```
In [9]: plt.xlabel('Angle [rad]',
  fontsize = 16)
```

```
In [10]:
```



Plotting

```
In [3]: import numpy as np
```

```
In [4]: import matplotlib.pyplot
  as plt
```

```
In [5]: x =
  np.linspace(0, 2 * np.pi, 100)
```

```
In [6]: plt.plot(x, np.sin(x), '*')
```

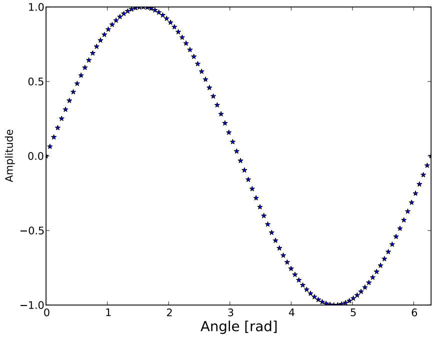
```
In [7]: plt.xlim(0, 10)
Out[7]: (0, 10)
```

```
In [8]: plt.xlim(0, 2 * np.pi)
Out[8]: (0, 6.2831853071795862)
```

```
In [9]: plt.xlabel('Angle [rad]',
  fontsize = 16)
```

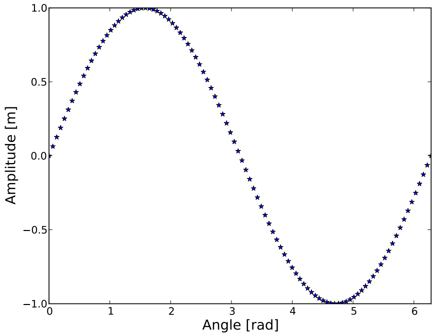
```
In [10]: plt.ylabel('Amplitude')
```

```
In [11]:
```



Plotting

```
In [3]: import numpy as np
In [4]: import matplotlib.pyplot
  as plt
In [5]: x =
  np.linspace(0, 2 * np.pi, 100)
In [6]: plt.plot(x, np.sin(x), '*')
In [7]: plt.xlim(0, 10)
Out[7]: (0, 10)
In [8]: plt.xlim(0, 2 * np.pi)
Out[8]: (0, 6.2831853071795862)
In [9]: plt.xlabel('Angle [rad]',
  fontsize = 16)
In [10]: plt.ylabel('Amplitude')
In [11]: plt.ylabel('Amplitude [m]',
  fontsize = 16)
In [12]:
```



Plotting

```
In [3]: import numpy as np
```

```
In [4]: import matplotlib.pyplot
  as plt
```

```
In [5]: x =
  np.linspace(0, 2 * np.pi, 100)
```

```
In [6]: plt.plot(x, np.sin(x), '*')
```

```
In [7]: plt.xlim(0, 10)
Out[7]: (0, 10)
```

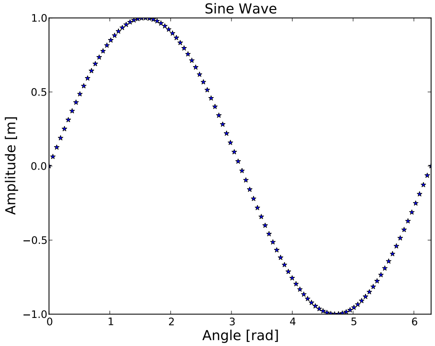
```
In [8]: plt.xlim(0, 2 * np.pi)
Out[8]: (0, 6.2831853071795862)
```

```
In [9]: plt.xlabel('Angle [rad]',
  fontsize = 16)
```

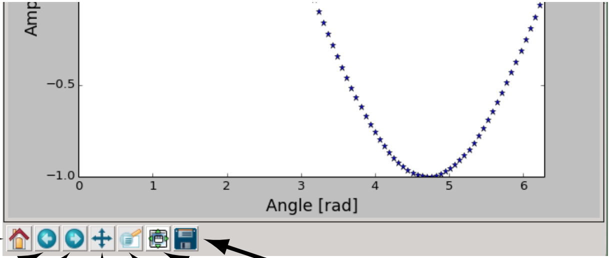
```
In [10]: plt.ylabel('Amplitude')
```

```
In [11]: plt.ylabel('Amplitude [m]',
  fontsize = 16)
```

```
In [12]: plt.title('Sine Wave',
  fontsize = 16)
```



What are those buttons?



Return to where you started.

Go back a step.

Go forward a step.

Grab the plot and move it around.

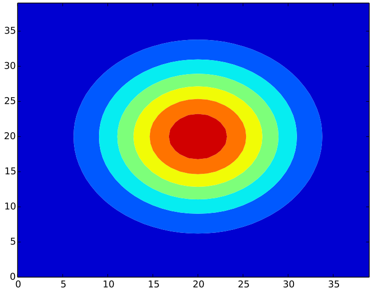
Zoom in on a section.

Adjust spacing between plots.

Save the figure.

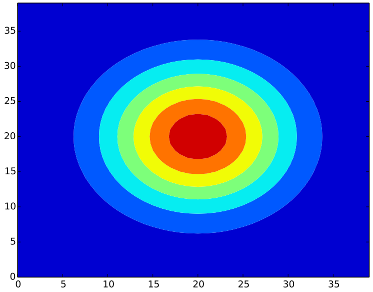
Pop quiz!

Using the numpy "mgrid" function, and the pylab "contourf" function, reproduce this plot of a Gaussian: $\boldsymbol{f(x,y) = exp(-(x^2+y^2))}$, for $\boldsymbol{-2 \le x \le 2}$ and $\boldsymbol{-2 \le y \le 2}$.



Pop quiz!

Using the numpy "mgrid" function, and the pylab "contourf" function, reproduce this plot of a Gaussian: $f(x, y) = exp(-(x^2+y^2))$, for $-2 \leq x \leq 2$ and $-2 \leq y \leq 2$.



```
In [16]: x, y = np.mgrid[-2:2:0.1, -2:2:0.1]
In [17]: g = exp(-(x**2 + y**2))
In [18]: contourf(g)
```

Statistics

SciPy contains all of the statistical functions that you'll probably ever need.

- The scipy.stats module is based around the idea of a 'random variable' type.
- A whole variety of standard distributions are available:
  - Continuous distributions: Normal, Maxwell, Cauchy, Chi-squared, Gumbel Left-scewed, Gilbrat, Nakagami, ...
  - Discrete distributions: Poisson, Binomial, Geometric, Bernoulli, ...
- The 'random variables' have all of the statistical properties of the distributions built into them already: cdf, pdf, mean, variance, moments, ...

Statistics, continued

```
In [19]:
```

```
In [19]:
```

Statistics, continued

```
In [19]:
```

```
In [19]: from scipy.stats
  import norm
```

```
In [20]:
```

Statistics, continued

```
In [19]:
```

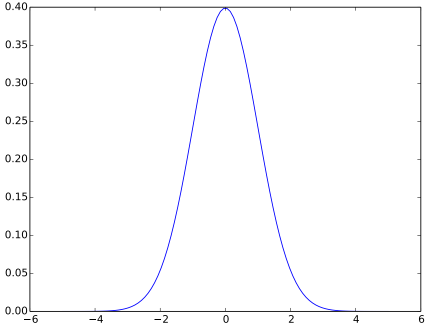
```
In [19]: from scipy.stats
  import norm
```

```
In [20]: x = linspace(-5, 5, 100)
```

```
In [21]:
```

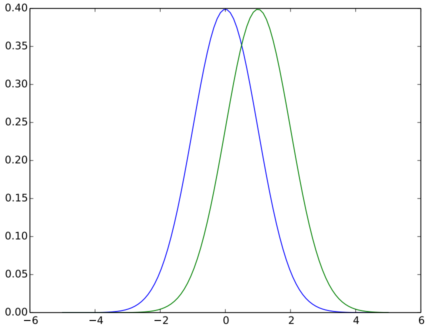
Statistics, continued

```
In [19]:
In [19]: from scipy.stats
  import norm
In [20]: x = linspace(-5, 5, 100)
In [21]: plot(x, norm.pdf(x))
In [22]:
```



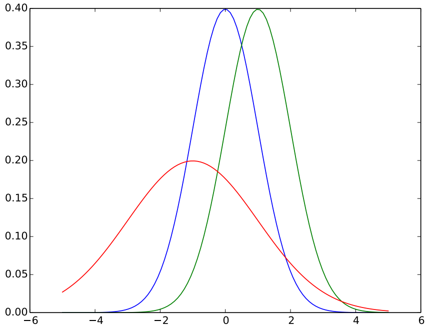
Statistics, continued

```
In [19]:
In [19]: from scipy.stats
  import norm
In [20]: x = linspace(-5, 5, 100)
In [21]: plot(x, norm.pdf(x))
In [22]: plot(x,
  norm.pdf(x, loc = 1))
In [23]:
```



Statistics, continued

```
In [19]:
In [19]: from scipy.stats
  import norm
In [20]: x = linspace(-5, 5, 100)
In [21]: plot(x, norm.pdf(x))
In [22]: plot(x,
  norm.pdf(x, loc = 1))
In [23]: plot(x,
  norm.pdf(x, loc = -1, scale = 2))
In [24]:
```



All continuous distributions take "loc" and "scale" as keyword parameters to adjust the location and scale of the distribution. In general the distribution of a random variable X is obtained from $(X - loc)/scale$. The default values are $loc = 0$ and $scale = 1$.

Statistics, continued more

```
In [24]:
```

Statistics, continued more

```
In [24]: from pylab import hist
```

```
In [25]:
```

Statistics, continued more

```
In [24]: from pylab import hist
```

```
In [25]: norm.mean(loc = -1,
  scale = 2)
Out[25]: -1.0
```

```
In [26]:
```

Statistics, continued more

```
In [24]: from pylab import hist
```

```
In [25]: norm.mean(loc = -1,
  scale = 2)
Out[25]: -1.0
```

```
In [26]: norm.std(loc = -1,
  scale = 2)
Out[27]: 2.0
```

```
In [28]:
```

Statistics, continued more

```
In [24]: from pylab import hist
```

```
In [25]: norm.mean(loc = -1,
  scale = 2)
Out[25]: -1.0
```

```
In [26]: norm.std(loc = -1,
  scale = 2)
Out[27]: 2.0
```

```
In [28]: norm.moment(3, loc = -1,
  scale = 2)
Out[28]: -13.0
```

```
In [29]:
```

Statistics, continued more

```
In [24]: from pylab import hist
```

```
In [25]: norm.mean(loc = -1,
  scale = 2)
Out[25]: -1.0
```

```
In [26]: norm.std(loc = -1,
  scale = 2)
Out[27]: 2.0
```

```
In [28]: norm.moment(3, loc = -1,
  scale = 2)
Out[28]: -13.0
```

```
In [29]: samples = norm.rvs(
  size = 1000, loc = -1,
  scale = 2)
```

```
In [30]:
```

Statistics, continued more

```
In [24]: from pylab import hist
```

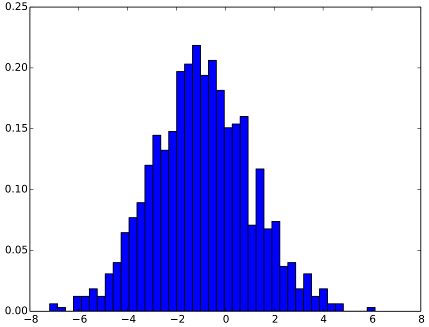
```
In [25]: norm.mean(loc = -1,
  scale = 2)
Out[25]: -1.0
```

```
In [26]: norm.std(loc = -1,
  scale = 2)
Out[27]: 2.0
```

```
In [28]: norm.moment(3, loc = -1,
  scale = 2)
Out[28]: -13.0
```

```
In [29]: samples = norm.rvs(
  size = 1000, loc = -1,
  scale = 2)
```

```
In [30]: h = hist(samples,
  bins = 41, normed = True)
```



```
In [31]:
```

Statistics, continued more

```
In [24]: from pylab import hist
```

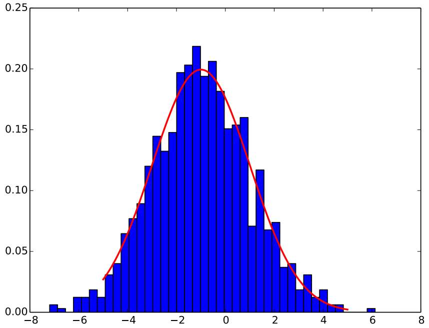
```
In [25]: norm.mean(loc = -1,
  scale = 2)
Out[25]: -1.0
```

```
In [26]: norm.std(loc = -1,
  scale = 2)
Out[27]: 2.0
```

```
In [28]: norm.moment(3, loc = -1,
  scale = 2)
Out[28]: -13.0
```

```
In [29]: samples = norm.rvs(
  size = 1000, loc = -1,
  scale = 2)
```

```
In [30]: h = hist(samples,
  bins = 41, normed = True)
```



```
In [31]: plot(x, norm.pdf(x,
    loc = -1, scale = 2), 'r',
    linewidth = 2)
```

Setting the seed

Sometimes you need consistency in your randomness:

- Pseudo-random numbers are generated from an initial 'seed'.
- This seed generates the first number, which is then used as the seed for the second number.
- If you need consistency in your random numbers (for debugging, for example), you can set the seed explicitly so that you get the same random numbers every time.
- Don't do this for production!

```
In [24]:

In [24]: norm.rvs()
Out[24]: 1.74481176421648

In [25]: norm.rvs()
Out[25]: -0.7612069008951028

In [26]:

In [26]: numpy.random.seed(1)

In [27]: norm.rvs()
Out[27]: 1.6243453636632417

In [28]:

In [28]: numpy.random.seed(1)

In [29]: norm.rvs()
Out[29]: 1.6243453636632417

In [30]:

In [30]: random.seed(1)

In [31]: norm.rvs()
Out[31]: 1.6243453636632417

In [32]:
```

Statistics, a discrete example

```
In [32]:
```

Note that discrete distributions have Probability Mass Functions (PMF) instead of Probability Distribution Functions (PDF).

Statistics, a discrete example

```
In [32]: from scipy.stats import
  poisson
In [33]:
```

Note that discrete distributions have Probability Mass Functions (PMF) instead of Probability Distribution Functions (PDF).

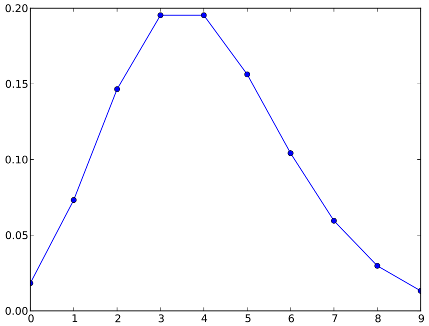
Statistics, a discrete example

```
In [32]: from scipy.stats import
  poisson
In [33]: x = arange(10)
In [34]:
```

Note that discrete distributions have Probability Mass Functions (PMF) instead of Probability Distribution Functions (PDF).

Statistics, a discrete example

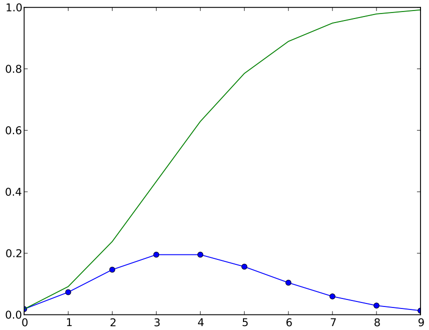
```
In [32]: from scipy.stats import
  poisson
In [33]: x = arange(10)
In [34]: plot(x,
  poisson.pmf(x, 4), 'o-')
In [35]:
```



Note that discrete distributions have Probability Mass Functions (PMF) instead of Probability Distribution Functions (PDF).

Statistics, a discrete example

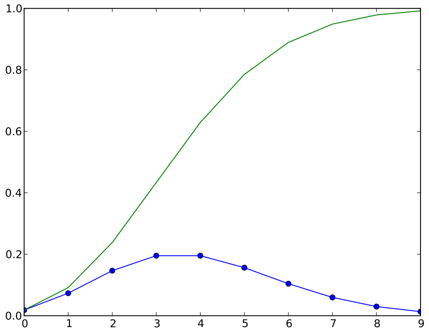
```
In [32]: from scipy.stats import
  poisson
In [33]: x = arange(10)
In [34]: plot(x,
  poisson.pmf(x, 4), 'o-')
In [35]: plot(x,
  poisson.cdf(x, 4)
In [36]:
```



Note that discrete distributions have Probability Mass Functions (PMF) instead of Probability Distribution Functions (PDF).

Statistics, a discrete example

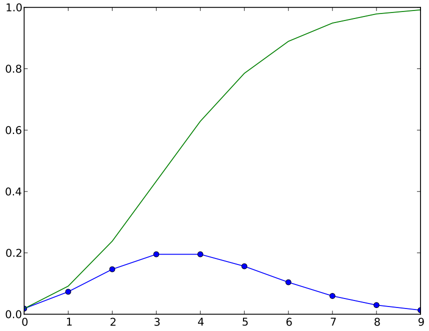
```
In [32]: from scipy.stats import
  poisson
In [33]: x = arange(10)
In [34]: plot(x,
  poisson.pmf(x, 4), 'o-')
In [35]: plot(x,
  poisson.cdf(x, 4)
In [36]: poisson.mean(4)
Out[36]: 4.0
In [37]:
```



Note that discrete distributions have Probability Mass Functions (PMF) instead of Probability Distribution Functions (PDF).

Statistics, a discrete example

```
In [32]: from scipy.stats import
  poisson
In [33]: x = arange(10)
In [34]: plot(x,
  poisson.pmf(x, 4), 'o-')
In [35]: plot(x,
  poisson.cdf(x, 4)
In [36]: poisson.mean(4)
Out[36]: 4.0
In [37]: poisson.var(4)
Out[37]: 4.0
```



Note that discrete distributions have Probability Mass Functions (PMF) instead of Probability Distribution Functions (PDF).

Polynomial fitting

```
In [38]:
```

Polynomial fitting

```
In [38]: from numpy import arange, zeros
  polyfit, polyval, random
```

```
In [39]:
```

Polynomial fitting

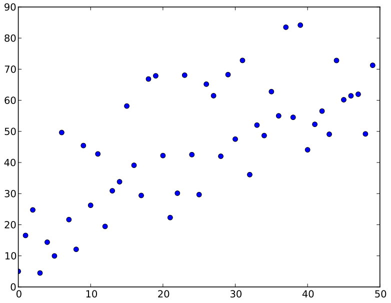
```
In [38]: from numpy import arange, zeros
  polyfit, polyval, random
In [39]: x = arange(50.)
In [40]:
```

Polynomial fitting

```
In [38]: from numpy import arange, zeros
  polyfit, polyval, random
In [39]: x = arange(50.)
In [40]: y = x + 50.0 * random.rand(50)
In [41]:
```

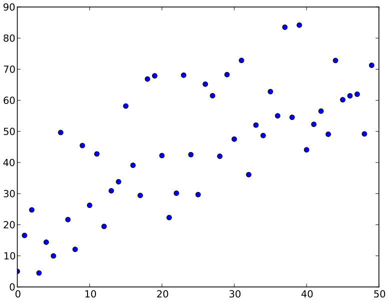
Polynomial fitting

```
In [38]: from numpy import arange, zeros
  polyfit, polyval, random
In [39]: x = arange(50.)
In [40]: y = x + 50.0 * random.rand(50)
In [41]: plot(x, y, 'o')
In [42]:
```



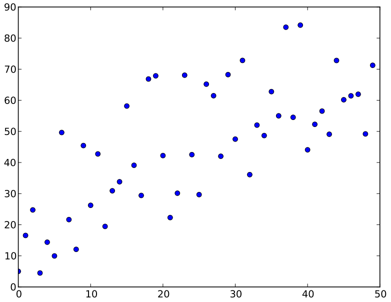
Polynomial fitting

```
In [38]: from numpy import arange, zeros
  polyfit, polyval, random
In [39]: x = arange(50.)
In [40]: y = x + 50.0 * random.rand(50)
In [41]: plot(x, y, 'o')
In [42]: fit = polyfit(x, y, 1)
In [43]:
```



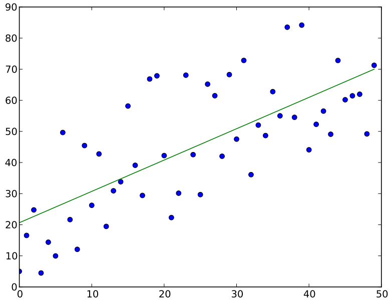
Polynomial fitting

```
In [38]: from numpy import arange, zeros
  polyfit, polyval, random
In [39]: x = arange(50.)
In [40]: y = x + 50.0 * random.rand(50)
In [41]: plot(x, y, ’o’)
In [42]: fit = polyfit(x, y, 1)
In [43]: fit
Out[43]: array([ 1.007358, 20.6469503])
In [44]:
```



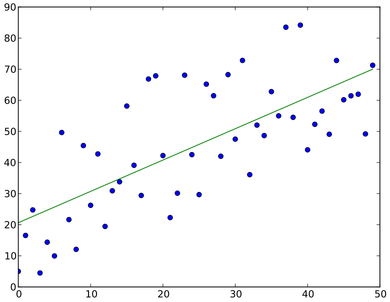
Polynomial fitting

```
In [38]: from numpy import arange, zeros
  polyfit, polyval, random
In [39]: x = arange(50.)
In [40]: y = x + 50.0 * random.rand(50)
In [41]: plot(x, y, 'o')
In [42]: fit = polyfit(x, y, 1)
In [43]: fit
Out[43]: array([ 1.007358, 20.6469503])
In [44]: plot(x, polyval(fit, x))
In [45]:
```



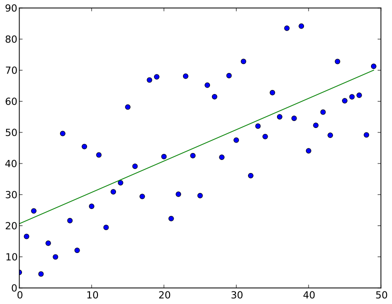
Polynomial fitting

```
In [38]: from numpy import arange, zeros
  polyfit, polyval, random
In [39]: x = arange(50.)
In [40]: y = x + 50.0 * random.rand(50)
In [41]: plot(x, y, 'o')
In [42]: fit = polyfit(x, y, 1)
In [43]: fit
Out[43]: array([ 1.007358, 20.6469503])
In [44]: plot(x, polyval(fit, x))
In [45]: fit = polyfit(x, y, 2)
In [46]:
```



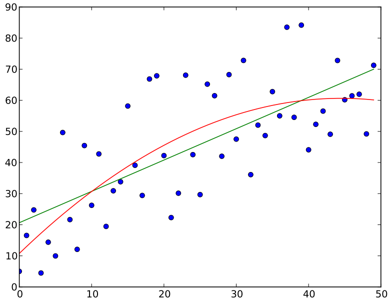
Polynomial fitting

```
In [38]: from numpy import arange, zeros
  polyfit, polyval, random
In [39]: x = arange(50.)
In [40]: y = x + 50.0 * random.rand(50)
In [41]: plot(x, y, 'o')
In [42]: fit = polyfit(x, y, 1)
In [43]: fit
Out[43]: array([ 1.007358, 20.6469503])
In [44]: plot(x, polyval(fit, x))
In [45]: fit = polyfit(x, y, 2)
In [46]: fit
Out[46]: array([ -0.02520835,
  2.24256777, 10.76527546])
In [47]:
```



Polynomial fitting

```
In [38]: from numpy import arange, zeros
  polyfit, polyval, random
In [39]: x = arange(50.)
In [40]: y = x + 50.0 * random.rand(50)
In [41]: plot(x, y, 'o')
In [42]: fit = polyfit(x, y, 1)
In [43]: fit
Out[43]: array([ 1.007358, 20.6469503])
In [44]: plot(x, polyval(fit, x))
In [45]: fit = polyfit(x, y, 2)
In [46]: fit
Out[46]: array([ -0.02520835,
  2.24256777, 10.76527546])
In [47]: plot(x, polyval(fit, x))
In [48]:
```



Dealing with inhomogeneous data

Not all data comes in an array format.

- Some data is organized more like an Excel file (tabular format) than as a numeric array.
- This means there are usually 'headers' (column titles), and sometimes indexes (row numbers).
- Generally speaking, the data contains a mixture of datatypes (strings, floats, integers, booleans), not just a single type.
- The data is commonly stored several different formats. We will touch on the CSV ("Comma Separated Values").

For this we will use the Python 'pandas' package, which allows us to use data frames (very similar to data frames in R).

Working with data frames

```
In [48]: import pandas as pd
In [49]: df = pd.read_csv("http://www.scinethpc.ca/~ejspence/311-reqs.csv")
In [50]:
In [50]: df.shape
Out[50]: (10000, 52)
In [51]:
In [51]: df.columns
Index([u'Unique Key', u'Created Date', u'Closed Date', u'Agency',
⋮
u'Ferry Direction', u'Ferry Terminal Name', u'Latitude', u'Longitude',
u'Location'], dtype='object')
In [52]:
```

The data have been cast into a 'DataFrame' type, which is the type used by pandas.

Working with data frames, continued

```
In [52]:
In [52]: df.values[0]
array([26589651, '10/31/2013 02:08:41 AM', nan, 'NYPD', 'New York
City Police Department', 'Noise - Street/Sidewalk', 'Loud Talking',
'Street/Sidewalk', 11432.0, '90-03 169 STREET', '169 STREET', '90 AVENUE',
'91 AVENUE', nan, nan, 'ADDRESS', 'JAMAICA', nan, 'Precinct', 'Assigned',
'10/31/2013 10:08:41 AM', '10/31/2013 02:35:17 AM', '12 QUEENS', 'QUEENS',
1042027.0, 197389.0, 'Unspecified', 'QUEENS', 'Unspecified', 'Unspecified',
'Unspecified', 'Unspecified', 'Unspecified', 'Unspecified', 'Unspecified',
'Unspecified', 'Unspecified', 'N', nan, nan, nan, nan, nan, nan, nan, nan,
nan, nan, nan, 40.70827532593202, -73.79160395779721, '(40.70827532593202,
-73.79160395779721)'], dtype=object)
In [53]:
```

Specifying the index gives us all the values in that row.

Noise-complaint data

Suppose we're only interested in the noise-complaint data.

```
In [53]: noise = df[df["Complaint Type"] == "Noise - Street/Sidewalk"]
```

```
In [54]: noise[0:3]
Out[54]:
      Unique Key             Created Date             Closed Date   Agency   \
 0      26589651   10/31/2013 02:08:41 AM                     NaN     NYPD
 16     26594086   10/31/2013 12:54:03 AM  10/31/2013 02:16:39 AM     NYPD
 25     26591573   10/31/2013 12:35:18 AM  10/31/2013 02:41:35 AM     NYPD

                          Agency Name           Complaint Type
 0    New York City Police Department  Noise - Street/Sidewalk  ...
 16   New York City Police Department  Noise - Street/Sidewalk
 25   New York City Police Department  Noise - Street/Sidewalk

                                          ⋮
```

```
In [55]: noise.shape
(71, 52)
```

This picks out all "Noise - Street/Sidewalk" complaints.

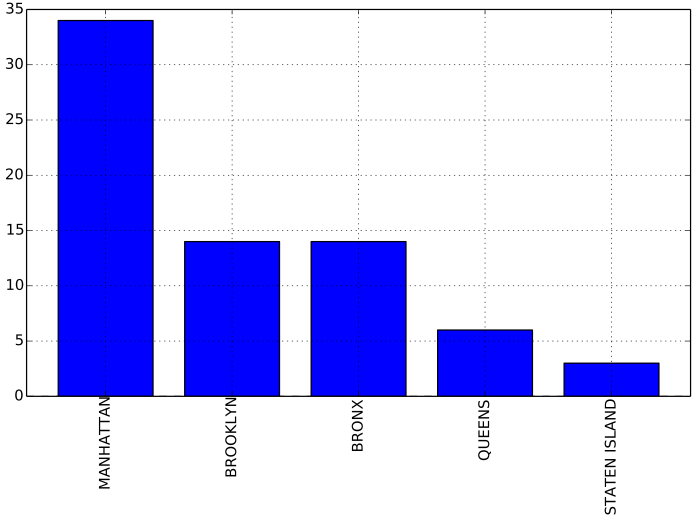
Who complains the most?

So which borough is responsible for the most complaints?

```
In [56]:

In [56]: noise["Borough"].value_counts()
 MANHATTAN       34
 BROOKLYN        14
 BRONX           14
 QUEENS           6
 STATEN ISLAND    3
 dtype: int64

In [57]:

In [57]: noise["Borough"].value_counts().plot(kind = "bar")

In [58]:
```

Manhattan!

Who complains the most?



Pop quiz!

Which borough submits the largest fraction of complaints, overall?

Pop quiz!

Which borough submits the largest fraction of complaints, overall?

```
In [60]:

In [60]: total = df.shape[0]

In [61]:

In [61]: df["Borough"].value_counts() / total
Out[61]:
 BROOKLYN         0.2984
 QUEENS           0.2114
 MANHATTAN        0.2040
 BRONX            0.1766
 Unspecified      0.0693
 STATEN ISLAND    0.0403
 dtype: float64

In [62]:
```

Brooklyn!

A note on file formats

CSV (Comma Separated Value) – or any text-based format – is the worst possible format for quantitative data. It manages the trifecta of being:

- Slow to read.
- Huge in size.
- Inaccurate.

Converting floating point numbers back and forth between internal representations and strings is slow and prone to trunctation errors.

Use binary formats whenever possible. Python has a few built-in formats that we'll look at. Portable formats like HDF5 (for data frames) or NetCDF4 (for matrices and arrays) are compact, accurate, fast (though not as fast as the built-in Python formats), portable, and can be read by tools other than Python.

Pickle

A flexible and useful Python format, pickle:

- Base64 encoding using readable ASCII
- Portable for the same version of python.
- In the pickle module.
- Flexible, can serialize almost any structure.

```
In [62]: import pickle
In [63]:
In [63]: a = zeros([1000, 1000])
In [64]:
In [64]: f = open('a.pickle', 'w')
In [65]: pickle.dump(a, f)
In [66]: f.close()
In [67]:
In [67]: g = open('a.pickle', 'r')
In [68]: b = pickle.load(g)
In [69]: g.close()
In [70]:
In [70]: b.shape
Out[70]: (1000, 1000)
In [71]:
```

Shelve

Shelve is built on top of pickle, but has some advantages:

- Your file handle is treated like a dictionary.
- As such, you can save as many different pieces of data as you like.
- And you can search the file for the data that you're after.
- Based on pickle, and thus has restrictions based on serialization.

```
In [71]: import shelve
In [72]:
In [72]: a = zeros([1000, 1000])
In [73]:
In [73]: f = shelve.open('my.data')
In [74]: f['a'] = a
In [75]: f.close()
In [76]:
In [76]: g = shelve.open('my.data')
In [77]: g.keys()
Out[77]: ['a']
In [78]: g['a'].shape
Out[78]: (1000, 1000)
In [79]:
```

Pass by value or reference?

Does Python "pass by value" or by "pass by reference"? (These are also sometimes called by "call by value" and "call by reference".)

Meaning, if I call a function, f(x), what exactly is 'x', when I get inside the function?

Pass by value or reference?

Does Python "pass by value" or by "pass by reference"? (These are also sometimes called by "call by value" and "call by reference".)

Meaning, if I call a function, f(x), what exactly is 'x', when I get inside the function?

- In pass-by-value languages (R, C++, Java), the value of x is copied and passed to the inside of the function. This can lead to memory problems if x is large.

Pass by value or reference?

Does Python "pass by value" or by "pass by reference"? (These are also sometimes called by "call by value" and "call by reference".)

Meaning, if I call a function, f(x), what exactly is 'x', when I get inside the function?

- In pass-by-value languages (R, C++, Java), the value of x is copied and passed to the inside of the function. This can lead to memory problems if x is large.
- In pass-by-reference languages (Fortran), the memory-address of x is passed to the inside of the function. As such, the function can modify the value of the variable and those changes will persist after the function is finished.

Pass by value or reference?

Does Python "pass by value" or by "pass by reference"? (These are also sometimes called by "call by value" and "call by reference".)

Meaning, if I call a function, f(x), what exactly is 'x', when I get inside the function?

- In pass-by-value languages (R, C++, Java), the value of x is copied and passed to the inside of the function. This can lead to memory problems if x is large.
- In pass-by-reference languages (Fortran), the memory-address of x is passed to the inside of the function. As such, the function can modify the value of the variable and those changes will persist after the function is finished.

Which is Python? The answer is: technically neither.
See the next slide.

Copying array variables

Use caution when copying array variables. There's a 'feature' here that may be unexpected.

```
In [79]: a = 10
In [80]: b = a
In [81]: a = 20
In [82]: a, b
Out[82]: (20, 10)
In [83]: a = array([[1,2,3], [2,3,4]])
In [84]: b = a
In [85]: a[1,0] = -10
In [86]: a
Out[86]:
 array([[ 1,  2,  3],
       [-10,  3,  4]])
In [87]:
```

```
In [87]:
In [87]: b
Out[87]:
 array([[ 1,  2,  3],
       [-10,  3,  4]])
In [88]: b = a.copy()
In [89]: a[1,0] = 16
In [90]: a
Out[90]:
 array([[ 1,  2,  3],
        [16,  3,  4]])
In [91]: b
Out[91]:
 array([[ 1,  2,  3],
       [-10,  3,  4]])
In [92]:
```

Copying array variables, continued

What happened here?

- Variables are all "pointers". They point to the value in memory (sort of).
- Integers are "immutable", which means their values can't be changed. When "b = a; a = 20", 'b' points to 10, and then 'a' points to a new integer, 20.
- Arrays are "mutable", meaning their values can be changed. The pointers 'a' and 'b' both point to the same object.

```
In [92]: a = 10
In [93]: b = a
In [94]: a = 20
In [95]: a, b
Out[95]: (20, 10)
In [96]: a = array([[1,2,3], [2,3,4]])
In [97]: b = a
In [98]: a[1,0] = -10
In [99]: a
Out[99]:
 array([[ 1,  2,  3],
        [-10, 3,  4]])
In [100]:
In [100]: b
Out[100]:
 array([[ 1,  2,  3],
        [-10, 3,  4]])
In [101]:
```

"Pass by object reference"

Technically, Python is "pass by object reference" (also called "call by sharing"):

- The inside of the function, f(x), (sort of) gets a *copy of a pointer* to the object 'x' (call the pointer 'y').
- If 'x' is immutable (number, string, boolean, tuple, ...), meaning the value can't be changed, and our pointer 'y' is reassigned within the function, then
  - a new object is created and 'y' points to that.
  - when the function exits 'y' disappears, and nothing changes outside the function.
- if 'x' is mutable (list, array, dictionary, ...), and our pointer 'y' changes what it points to (assignment, append, expansion, whatever), then 'x' itself is also modified.

For most intents and purposes, you should think of Python as pass-by-reference.

Loop indices

Which is faster? Why?

```
# mylooper1.py
from numpy import zeros

a = zeros([20000, 20000])
for i in xrange(20000):
  for j in xrange(20000):
    a[i,j] = i + j
```

```
# mylooper2.py
from numpy import zeros

a = zeros([20000, 20000])
for j in xrange(20000):
  for i in xrange(20000):
    a[i,j] = i + j
```

Loop indices

Which is faster? Why?

Python is *row major*. This means that, in memory, the array is stored in blocks of constant row. We've seen this on previous slides when we declare an array:

```
In [101]: a = array([[1,2,3], [2,3,4]])
In [102]: a
Out[102]:
 array([[1, 2, 3],
        [2, 3, 4]])
```

```
# mylooper1.py
from numpy import zeros

a = zeros([20000, 20000])
for i in xrange(20000):
  for j in xrange(20000):
    a[i,j] = i + j
```

```
# mylooper2.py
from numpy import zeros

a = zeros([20000, 20000])
for j in xrange(20000):
  for i in xrange(20000):
    a[i,j] = i + j
```

Always, always loop over the right-most index on the inner-most loop. These array elements will closest together in memory, and thus will be accessed faster within the loop.

Iterators and generators

Why did I use "xrange" instead of "range"? What's the difference?

```
# mylooper1.py
from numpy import zeros

a = zeros([20000, 20000])
for i in xrange(20000):
  for j in xrange(20000):
    a[i,j] = i + j
```

```
In [101]:
```

Iterators and generators

Why did I use "xrange" instead of "range"? What's the difference?

- "xrange" is like an "iterator". It only returns values on demand.

```
# mylooper1.py
from numpy import zeros

a = zeros([20000, 20000])
for i in xrange(20000):
  for j in xrange(20000):
    a[i,j] = i + j
```

```
In [101]: xrange(int(1e10))
Out[101]: xrange(10000000000)
In [102]:
```

Iterators and generators

Why did I use "xrange" instead of "range"? What's the difference?

- "xrange" is like an "iterator". It only returns values on demand.
- "range" returns a list, which requires all the memory up front.

```
# mylooper1.py
from numpy import zeros

a = zeros([20000, 20000])
for i in xrange(20000):
  for j in xrange(20000):
    a[i,j] = i + j
```

```
In [101]: xrange(int(1e10))
Out[101]: xrange(10000000000)
```

```
In [102]: range(int(1e10))
---------------------------------------------------------------------------
MemoryError                               Traceback (most recent call last)
<ipython-input-19-2db10653fd33> in <module>()
----> 1 range(int(1e10))
MemoryError:
```

```
In [103]:
```

Building your own generators

If you create a function which returns a list or array, it may be in your best (memory) interest to create a "generator" function:

- A generator function is 'lazily' evaluated: values are only calculated and returned when needed.
- More importantly: values are thrown away when they're no longer needed.
- This saves alot of memory.
- To create a generator function, use the 'yield' command to return just the single element that your want returned at each iteration.

```python
# mygen.py
def firstn1(n):
  nums = [0]
  for i in xrange(1,n):
    temp = nums[-1]
    nums.append(i + temp)
  return nums

def firstn2(n):
  temp = 0
  for i in xrange(n):
    temp += i
    yield temp
```

Building your own generators, continued

```
In [103]: from mygen import firstn1,
firstn2
```

```
In [104]: firstn1(3)
Out[104]: [0, 1, 3]
```

```
In [105]:
```

```
In [105]: firstn2(3)
Out[105]: <generator object firstn2 at
0x7f53f52ac7d0>
```

```
In [106]: list(firstn2(3))
Out[106]: [0, 1, 3]
```

```
In [107]: for i in firstn2(3):
....:     print i
0
1
3
```

```
In [107]:
```

```
# mygen.py
def firstn1(n):
  nums = [0]
  for i in xrange(1,n):
    temp = nums[-1]
    nums.append(i + temp)
  return nums

def firstn2(n):
  temp = 0
  for i in xrange(n):
    temp += i
    yield temp
```

Using 'list' forces the generator to generate all the values, and defeats the purpose of having a generator. This is for demonstration only.

Enough to get started

There's obviously alot more to learn about using Python. Nonetheless, this is enough to get you started.

We'll discuss implementing Python functions in parallel this afternoon.

- NumPy: `http://wiki.scipy.org/Tentative_NumPy_Tutorial`
- SciPy: `http://docs.scipy.org/doc/scipy/reference/tutorial`