First Steps with
Numerical Computing in Python

Paul M. Magwene
Spring 2016

How to use IPython notebooks

This document was written in an IPython notebook. IPython notebooks allow us to weave together explantory text, code, and figures.

Don't copy and paste!

Learning to program has similarities to learning a foreign language. You need to practice writing your own code, not just copying and pasting it from another document (that's why I'm providing this as a PDF rather than a notebook itself, to make the copy-and-paste process less convenient).

Part of the practice of learning to program is making mistakes (bugs). Learning to find and correct bugs in your own code is vital.

Code cells

Each of the grey boxes below that has `In [n]:` to the left shows a so called "code cell". The text in the code cells is what you should type into the code cells of your own notebook. The regions starting with `Out [n]:` show you the result of the code you type in the proceeding input cell(s).

Evaluating code cells

After you type Python code into a code cell, hit `Shift-Enter` (hold down the `Shift` key while you press the `Enter` (or `Return`) key) to evaluate the code cell. If you type valid code you'll usually get some sort of output (at least in these first few examples). If you make a mistake and get an error message, click the input code cell and try and correct your mistake(s).

Try your own code

Test your understanding of the examples I've provided by writing additional code to illustrate the same principle or concept. Don't be afraid to make mistakes.

Help and Documentation

A key skill for becoming an efficient programmer is learning to efficiently navigate documentation resources. The Python standard library is very well documented, and can be quickly accessed from the IPython notebook help menu or online at the http://python.org (http://python.org) website. Similar links to some of the more commonly used scientific and numeric libraries are also found in the Ipython help menu.

In addition, there are several ways to access abbreviated versions of the documentation from the interpetter itself.

In [1]:
```
help(min)
```

```
Help on built-in function min in module builtins:

min(...)
    min(iterable, *[, default=obj, key=func]) -> value
    min(arg1, arg2, *args, *[, key=func]) -> value

    With a single iterable argument, return its smallest item. The
    default keyword-only argument specifies an object to return if
    the provided iterable is empty.
    With two or more arguments, return the smallest argument.
```

In [2]:
```
?min # this will pop-up a documentation window in the ipython notebook
```

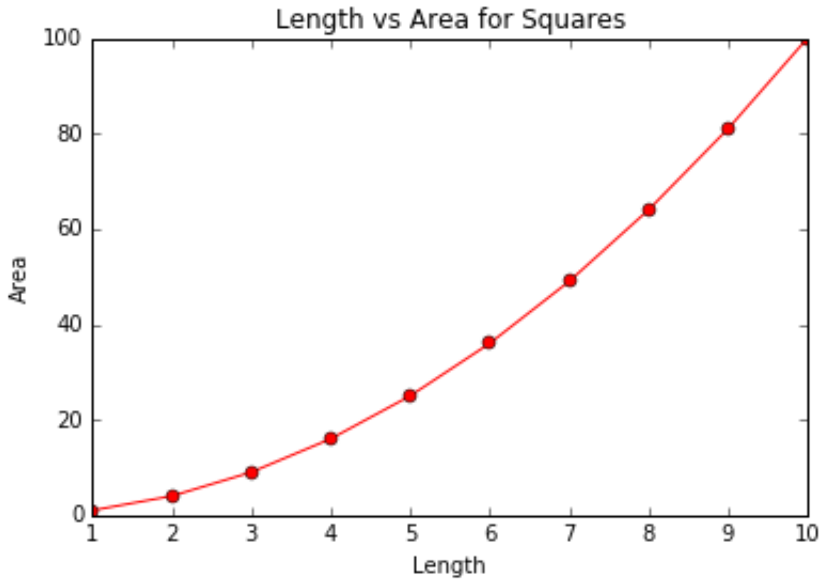
Gee-whiz!

Let's kick off our tour of Python with some nice visualizations. In this first section I'm not going to explain any of the code in detail, I'm simply going to generate some figures to show of some of what Python is capable of. However, once you work your way through this notebook you should be able to come back to this first section and understand most of the code written here.

In [3]:
```
%matplotlib inline
from numpy import *
from scipy import stats
from matplotlib.pyplot import *
```

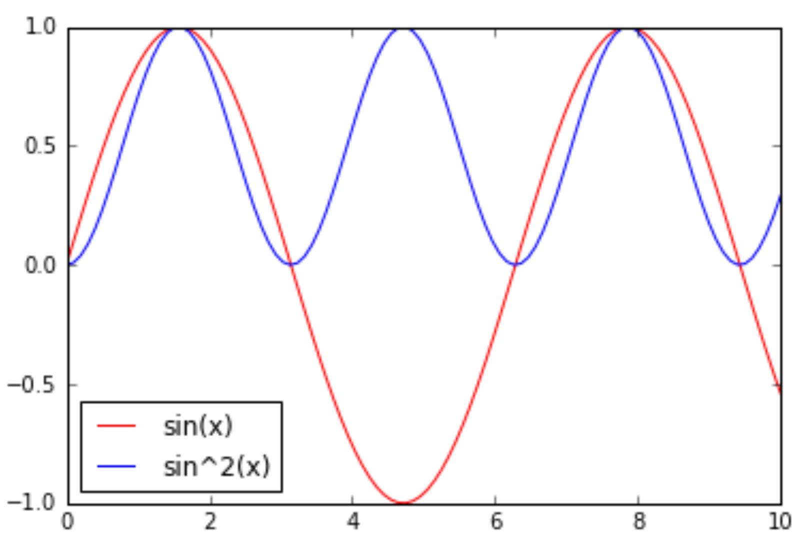
In [4]:

```
# this is a comment
x = array([1,2,3,4,5,6,7,8,9,10])
plot(x,x**2, color='red', marker='o')
xlabel("Length")
ylabel("Area")
title("Length vs Area for Squares")
pass
```



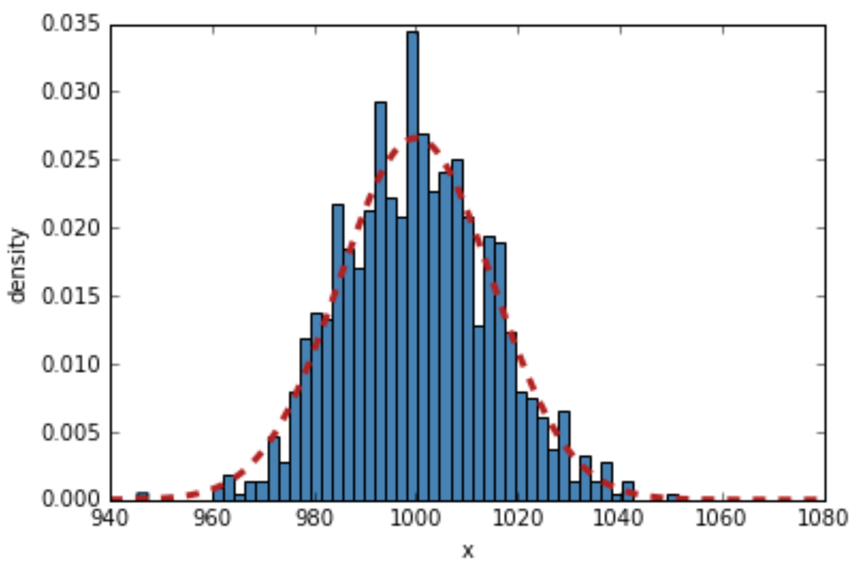
In [5]:

```
x = linspace(0, 10, 100) # generate 100 evenly space points
                         # between 0 and 10
sinx = sin(x)
sinsqrx = sinx * sinx

plot(x, sinx, color='red', label='sin(x)')
plot(x, sinsqrx, color='blue', label='sin^2(x)')
legend(loc='best')  # add optional legend to plot
pass
```



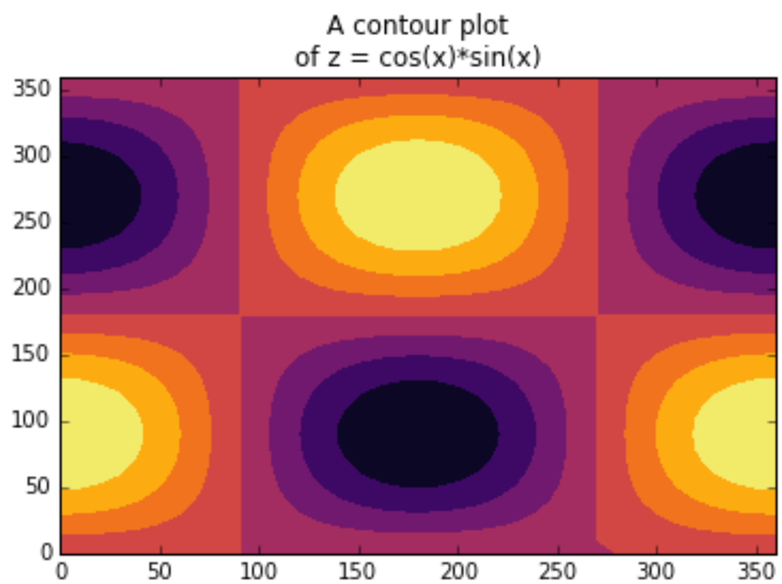
In [6]:

```
# draw 1000 random samples from a normal distribution
# with mean = 1000, sd = 15
mean = 1000
sd = 15
samples = random.normal(mean, sd, size=1000)

# draw a histogram
# normed means to make the total area under the
# histogram sum to 1 (i.e. a density histogram)
hist(samples, bins=50, normed=True, color='steelblue')

# draw probability density function for a normal
# distribution with the same parameters
x = linspace(940,1080,250)
y = stats.norm.pdf(x, loc=mean, scale=sd)
plot(x, y, color='firebrick', linestyle='dashed', linewidth=3)

# label axes
xlabel("x")
ylabel("density")

pass
```

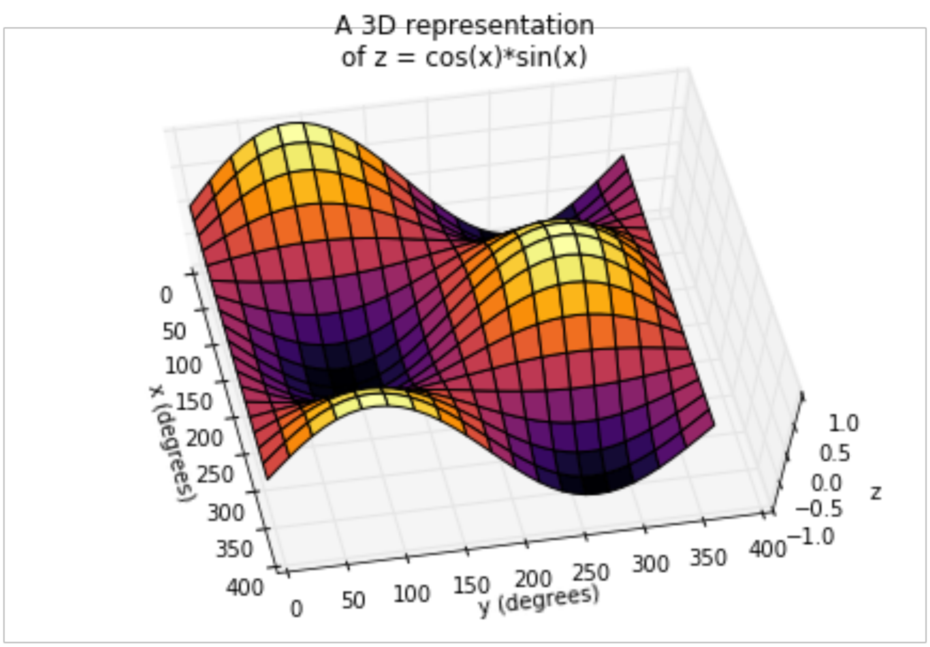


In [7]:

```
# the function of 2 variables we want to plot
def f(x,y):
    return cos(radians(x)) * sin(radians(y))

# generate a grid of x,y points at 10 step
# intervals from 0 to 360
x,y = meshgrid(arange(0, 361, 10), arange(0, 361, 10))

# calculate a function over the grid
z = f(x,y)

# draw a contour plot representing the function f(x,y)
contourf(x, y, z, cmap='inferno')

title("A contour plot\nof z = cos(x)*sin(x)")

pass
```



In [8]:
```
# function from previous plot, now represented in 3D
from mpl_toolkits.mplot3d import Axes3D

fig = figure()
ax = Axes3D(fig)
ax.plot_surface(x, y, z, rstride=2, cstride=2, cmap='inferno')

# setup axis labels
ax.set_xlabel("x (degrees)")
ax.set_ylabel("y (degrees)")
ax.set_zlabel("z")

# set elevation and azimuth for viewing
ax.view_init(68, -11)

title("A 3D representation\nof z = cos(x)*sin(x)")

pass
```



Numeric data types

One of the simplest ways to use the Python interpretter is as a fancy calculator. We'll illustrate this below and use this as an opportunity to introduce the core numeric data types that Python supports.

In [9]:
```
# this is a comment, the interpretter ignores it
# you can use comments to add short notes or explanation

2 + 10 # add two integers (whole numbers)
```

Out[9]: 12

In [10]:
```
2.0 + 10.0 # add two floating point numbers (real (decimal) numbers)
```
Out[10]: 12.0

In [11]:
```
2 + 10.0 # operations that mix integers and floats return floats
```
Out[11]: 12.0

In [12]:
```
2 * 10 # multiplication of integers
```
Out[12]: 20

In [13]:
```
2.0 * 10.0 # multiplication of floats
```
Out[13]: 20.0

In [14]:
```
1.0/5.0 # division
```
Out[14]: 0.2

In [15]:
```
2/5 # in Python 2 this used to default to integer division
    # in Python 3 division always returns a float
```
Out[15]: 0.4

In [16]:
```
10 % 3 # The % (modulo) operator yields the remainder after division
```
Out[16]: 1

In [17]:
```
2**10  # exponentiation -- 2 raised to the power 10
```
Out[17]: 1024

In [18]:
```
2**0.5  # exponentiation with fractional powers
        # **0.5 = square root, **(1/3.) = cube root
```
Out[18]: 1.4142135623730951

In [19]:
```
(10+2)/(4-5)  # numerical operators differ in their precedence
              # contrast the output of this line with the line below
```
Out[19]: -12.0

In [20]:
```
(10+2)/4-5   # it is a good habit to use parentheses to disambiguate
             # potentially confusing calculations
```
Out[20]: -2.0

In [21]:
```
(1 + 1j)  # complex numbers; we won't use these in the course
          # but you might occasionally find the need for them
          # in biological research
```

Out[21]: (1+1j)

In [22]:
```
(1 + 1j) + (3 + 2j) # adding complex numbers
```

Out[22]: (4+3j)

In [23]:
```
(0 + 1j) * (1j)  # complex multiplication
```

Out[23]: (-1+0j)

Querying objects for their type

There is a built-in Python function called `type` that we can use to query a variable for it's data type.

In [24]:
```
type(2)
```

Out[24]: int

In [25]:
```
type(2.0)
```

Out[25]: float

In [26]:
```
type(2 + 10.0)  # when adding variables of two numeric types, the outc
ome
                # is always the more general type
```

Out[26]: float

Booleans

Python has a data type to represent True and False values (Boolean variables) and supports standard Boolean operators like "and", "or", and "not"

In [27]:
```
x = True
y = False
```

In [28]:
```
x
```

Out[28]: True

In [29]:
```python
not x
```
Out[29]: False

In [30]:
```python
y
```
Out[30]: False

In [31]:
```python
not y # if True return False, if False return True
```
Out[31]: True

In [32]:
```python
x and y # if both arguments are True return true, else return False
```
Out[32]: False

In [33]:
```python
x and (not y)
```
Out[33]: True

In [34]:
```python
x or y  # if either argument is True, return True, else return False
```
Out[34]: True

Comparison operators

Python supports comparison operators on numeric data types. When you carry out a comparison you get back a Boolean (True,False) value.

In [35]:
```python
4 < 5 # less than
```
Out[35]: True

In [36]:
```python
4 > 5 # greater than
```
Out[36]: False

In [37]:
```python
5 <= 5.0 # less than or equal to
```
Out[37]: True

In [38]:
```python
5 == 5 # tests equality
```
Out[38]: True

In [39]:
```python
5 == (5**0.5)**2  # the results of this comparison might surprise you
```
Out[39]: False

In [40]:
```
(5**0.5)**2 # the problem is that sqrt(5) can not be represented
            # exactly with floating point numbers. This is not a
            # limitation of only Python but is generally true
            # for all programming languages
```

Out[40]: 5.000000000000001

In [41]:
```
# here's one way to test approximate equality when you suspect
# a floating point calculation might be imprecise

epsilon = 0.0000001
(5 - epsilon) <= ((5**0.5)**2) <= (5 + epsilon)
```

Out[41]: True

Variable assignment

A value or the result of a calculation can be given a name, and then reused in a different context by referring to that name. This is called variable assignment.

In [42]:
```
pi = 3.141592654
radius = 4.0
area_circ = pi * radius**2
# notice that you don't get any output from this code cell
```

In [43]:
```
# however, once you evaluate this code cell you will see
# the results of your calculation
area_circ
```

Out[43]: 50.265482464

Functions

A "function" is a named sequence of statements that performs a computation. Functions allow us to encapsulate or abstract away the steps required to perform a useful operation or calculation.

There are a number of Python funtions that are always available to you:

In [44]:
```
min(1,2) # find the minimum of its input
```

Out[44]: 1

In [45]:
```
max(10, 9, 11) # find maximum of inputs
```

Out[45]: 11

In [46]:
```
abs(-99)  # return absolute value of numerical input
```
Out[46]: 99

There are many other built-in functions, and we'll see more examples of these below. See the Python documentation on "Built-in Functions" (https://docs.python.org/3.5/library/functions.html) for more details.

Defining functions

You can write your own functions. The general form of a function definition in Python is:

```
def func_name(arg1, arg2, ...):
    body of function
    return result
```

Note:

- Python is white space sensitive, body of a function must be indented (idiomatic style is to indent by 4 spaces NOT tabs)
- Use a Python aware editor/environment to help get indenting correct. Jupyter will help you get the indentation correct

In [47]:
```
# a function that carries out a simple mathematical calculation

def area_of_circle(radius):
    """radius of circle --> area of circle"""
    return 3.141592654 * radius**2
```

In [48]:
```
area_of_circle(1)
```
Out[48]: 3.141592654

In [49]:
```
area_of_circle(8)
```
Out[49]: 201.061929856

Importing Functions

Python has a mechanism to allow you to build libraries of code, which can then be "imported" as needed. Python libraries are usually referred to as "modules".

Here’s how we would make functions and various definitions from the `math` module available for use.

In [50]:
```
import math
```

In [51]:
```
math.cos(2 * 3.141592654)  # cosine
```

Out[51]: 1.0

In [52]:
```
math.pi  # a constant defined in math
```

Out[52]: 3.141592653589793

In [53]:
```
pi = math.pi
```

In [54]:
```
math.cos(2 * pi)
```

Out[54]: 1.0

If you get tired of writing the module name, you can import all the functions from a module by writing `from math import *`. You have to be careful with this though, as any functions or constants imported this way wil overwrite any variables/names in your current environment that already exits.

At the beginning of this notebook I imported a library for numerical computing called NumPy (http://www.numpy.org) as well as a library for plotting called Matplotlib (http://matplotlib.org).

In [55]:
```
from numpy import *
from matplotlib.pyplot import *
```

Numpy includes most of the functions defined in the math module so we didn't really need to add the `math.` prefix.

In [56]:
```
exp(1) # e^1
```

Out[56]: 2.7182818284590451

In [57]:
```
log(e) # natural logarithm of e
```

Out[57]: 1.0

In [58]:
```
log10(100) # log base 10 of 100
```

Out[58]: 2.0

Lists

Lists are the simplest "data structure". Data structures are computational objects for storing, accessing, and operating on data.

List represent ordered collections of arbitrary objects. We'll begin by working with lists of numbers.

In [59]:
```
x = [1,2,3,4,5] # a list with the numbers 1..5
x
```

Out[59]: `[1, 2, 3, 4, 5]`

In [60]:
```
# a list with floats and ints and complex numbers
y = [2.0, 4, 6, 8, 10.0, 11, (1+1j), 3.14159]
y
```

Out[60]: `[2.0, 4, 6, 8, 10.0, 11, (1+1j), 3.14159]`

In [61]:
```
# lists of a length. We can use the `len` function to get this
len(x)
```

Out[61]: `5`

In [62]:
```
len(y)
```

Out[62]: `8`

Indexing lists

Accessing the elements of a list is called "indexing". In Python lists are "zero-indexed" which means when you can access lists elements, the first element has the index `0`, the second element has the index `1`, ..., and the last element has the index `len(x)-1`.

In [63]:
```
z = [2, 4, 6, 8, 10]
z[0] # first element
```

Out[63]: `2`

In [64]:
```
z[3]  # fourth element
```

Out[64]: `8`

In [65]:
```
len(z)
```

Out[65]: `5`

In [66]:
```python
z[5]  ## this generates an error -- why?
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent cal
l last)
<ipython-input-66-7557a87402ea> in <module>()
----> 1 z[5]  ## this generates an error -- why?

IndexError: list index out of range
```

In [67]:
```python
z[4] # last element of z
```

Out[67]:
```
10
```

You can use negative indexing to get elements from the end of a list.

In [68]:
```python
z[-1] # last element
```

Out[68]:
```
10
```

In [69]:
```python
z[-2] # second to last element
```

Out[69]:
```
8
```

Indexing can be used to get, set, and delete items in a list.

In [70]:
```python
m = [1, 2, 4, 6, 8, "hike"]
```

In [71]:
```python
m[-1] = "learning python is so great!"  # set the last element
m
```

Out[71]:
```
[1, 2, 4, 6, 8, 'learning python is so great!']
```

In [72]:
```python
del m[0]
m
```

Out[72]:
```
[2, 4, 6, 8, 'learning python is so great!']
```

You can append and delete list elements as well as concatenate two lists

In [73]:
```python
x = [1,2,3]
y = ['a', 'b', 'c', 'd']
```

In [74]:
```
x.append(4)
x
```

Out[74]: [1, 2, 3, 4]

In [75]:
```
x + y
```

Out[75]: [1, 2, 3, 4, 'a', 'b', 'c', 'd']

Slicing lists

Python lists support the notion of ‘slices’ - a continuous sublist of a larger list. The following code illustrates this concept.

In [76]:
```
c = ['a','b','c','d','e','f']
```

In [77]:
```
c[0:3]  # get the elements of from index 0 up to
        # but not including the element at index 3
```

Out[77]: ['a', 'b', 'c']

In [78]:
```
c[:3]  # same as above, first index implied
```

Out[78]: ['a', 'b', 'c']

In [79]:
```
c[2:5]  # from element 2 up to 5
```

Out[79]: ['c', 'd', 'e']

In [80]:
```
c[3:]  # from index three to end (last index implied)
```

Out[80]: ['d', 'e', 'f']

In [81]:
```
c[-1:0]  # how come this returned an empty list?
```

Out[81]: []

List slices support a "step" specified by a third colon

In [82]:
```
c[0:5:2]  # c from 0 to 5, step by 2
```

Out[82]: ['a', 'c', 'e']

In [83]:
```
# you can you a negative step to walk backward over a list
# note where the output stops (why didn't we get 'a'?)
c[-1:0:-1]
```

Out[83]: ['f', 'e', 'd', 'c', 'b']

As with single indexing, the slice notation can be used to set elements of a list.

In [84]:
```
c[2:4] = ['C', 'D']
c
```

Out[84]: ['a', 'b', 'C', 'D', 'e', 'f']

Finally, there are a number of useful methods associated with list objects, such as `reverse()` and `sort()`.

In [85]:
```
d = [1, 5, 3, 4, 1, 11, 3]
d.sort() # sort in place
d
```

Out[85]: [1, 1, 3, 3, 4, 5, 11]

In [86]:
```
d.reverse() # reverse in place
d
```

Out[86]: [11, 5, 4, 3, 3, 1, 1]

NumPy arrays

NumPy is an extension package for Python that provides many facilities for numerical computing. There is also a related package called SciPy that provides even more facilities for scientific computing. Both NumPy and SciPy can be downloaded from http://www.scipy.org/ (http://www.scipy.org/). NumPy does not come with the standard Python distribution, but it does come as an included package if you use the Anaconda Python distribution. The NumPy package comes with documentation and a tutorial. You can access the documentation here: http://docs.scipy.org/doc/ (http://docs.scipy.org/doc/).

The basic data structure in NumPy is the array, which you've already seen in several examples above. As opposed to lists, all the elements in a NumPy array must be of the same type (but this type can differ between different arrays). Arrays are commonly used to represent matrices (2D-arrays) but can be used to represent arrays of arbitrary dimension (n-dimensional arrays).

Arithmetic operations on NumPy arrays

In [87]:
```
from numpy import *
```

In [88]:
```
x = array([2,4,6,8,10])
x
```

Out[88]:
```
array([ 2,  4,  6,  8, 10])
```

In [89]:
```
type(x)
```

Out[89]:
```
numpy.ndarray
```

In [90]:
```
-x
```

Out[90]:
```
array([ -2,  -4,  -6,  -8, -10])
```

In [91]:
```
x**2
```

Out[91]:
```
array([  4,  16,  36,  64, 100])
```

In [92]:
```
x * pi
```

Out[92]:
```
array([  6.28318531,  12.56637061,  18.84955592,  25.13274123,  31.4
1592654])
```

Notice how all the arithmetic operations operate elementwise on arrays. You can also perform arithmetic operations between arrays, which also operate element wise

In [93]:
```
 y = array([0, 1, 3, 5, 9])
```

In [94]:
```
x + y
```

Out[94]:
```
array([ 2,  5,  9, 13, 19])
```

In [95]:
```
x * y
```

Out[95]:
```
array([ 0,  4, 18, 40, 90])
```

In [96]:
```
z = array([1, 4, 7, 11])
```

```
In [97]: x + z
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-97-f1bfe8206f1c> in <module>()
----> 1 x + z

ValueError: operands could not be broadcast together with shapes (5,) (4,)
```

The last example above shows that the lengths of the two arrays have to be the same in order to do element-wise operations.

By default,most operations on arrays work element-wise. However there are a variety of functions for doing array-wise operations such as matrix multiplication or matrix inversion. Here are a few examples of using NumPy arrays to represent matrices:

```
In [98]: m = np.array([[1,2],
                       [3,4]])
```

```
In [99]: m
```

```
Out[99]: array([[1, 2],
                [3, 4]])
```

```
In [100]: m.transpose()
```

```
Out[100]: array([[1, 3],
                 [2, 4]])
```

```
In [101]: linalg.inv(m)
```

```
Out[101]: array([[-2. ,  1. ],
                 [ 1.5, -0.5]])
```

Indexing and Slicing NumPy arrays

Like the built-in lists, NumPy arrays are zero-indexed.

```
In [102]: x
```

```
Out[102]: array([ 2,  4,  6,  8, 10])
```

In [103]:
```
x[0]
```
Out[103]:
```
2
```

In [104]:
```
x[1]
```
Out[104]:
```
4
```

In [105]:
```
x[4]
```
Out[105]:
```
10
```

In [106]:
```
x[5]
```
```
--------------------------------------------------------------------
-------
IndexError                                Traceback (most recent cal
l last)
<ipython-input-106-e8c2945f243d> in <module>()
----> 1 x[5]

IndexError: index 5 is out of bounds for axis 0 with size 5
```

Again, you can use negative indexing to get elements from the end of the vector and slicing to get subsets of the array.

In [107]:
```
x[-1]
```
Out[107]:
```
10
```

In [108]:
```
x[-2]
```
Out[108]:
```
8
```

In [109]:
```
x[2:]
```
Out[109]:
```
array([ 6,  8, 10])
```

In [110]:
```
x[::3] # every third element of x
```
Out[110]:
```
array([2, 8])
```

Comparison operators on arrays

NumPy arrays support the comparison operators, returning arrays of Booleans.

```
In [111]: x
```

```
Out[111]: array([ 2,  4,  6,  8, 10])
```

```
In [112]: x < 5
```

```
Out[112]: array([ True,  True, False, False, False], dtype=bool)
```

```
In [113]: x >= 6
```

```
Out[113]: array([False, False,  True,  True,  True], dtype=bool)
```

Combining indexing and comparison on arrays

NumPy arrays allows us to combine the comparison operators with indexing. This facilitates data filtering and subsetting.

```
In [114]: x = array([2, 4, 6, 10, 8, 7, 9, 2, 11])
```

```
In [115]: x[x > 5]
```

```
Out[115]: array([ 6, 10,  8,  7,  9, 11])
```

```
In [116]: x[x != 2]
```

```
Out[116]: array([ 4,  6, 10,  8,  7,  9, 11])
```

```
In [117]: x[logical_or(x <4, x > 8)]
```

```
Out[117]: array([ 2, 10,  9,  2, 11])
```

In the first example we retrieved all the elements of `x` that are larger than 5 (read "x where x is greater than 5"). In the second example we retrieved those elements of `x` that did not equal six. The third example is slightly more complicated. We combined the `logical_or` function with comparison and indexing. This allowed us to return those elements of the array `x` that are either less than four *or* greater than six. Combining indexing and comparison is a powerful concept. See the numpy documentation on logical functions (http://docs.scipy.org/doc/numpy/reference/routines.logic.html) for more information.

Generating Regular Sequences

Creating sequences of numbers that are separated by a specified value or that follow a particular patterns turns out to be a common task in programming. Python and NumPy have functions to simplify this task.

In [118]:
```
arange(10)
```

Out[118]:
```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

In [119]:
```
# generate numbers from 3 to 12 (non-inclusive) stepping by 4
arange(3, 12, 4)
```

Out[119]:
```
array([ 3,  7, 11])
```

In [120]:
```
arange(1,10,0.5)
```

Out[120]:
```
array([ 1. ,  1.5,  2. ,  2.5,  3. ,  3.5,  4. ,  4.5,  5. ,  5.5,  
6. ,
        6.5,  7. ,  7.5,  8. ,  8.5,  9. ,  9.5])
```

You can also do some fancy tricks on lists to generate repeating patterns.

In [121]:
```
[True,True,False]*3
```

Out[121]:
```
[True, True, False, True, True, False, True, True, False]
```

Mathematical functions applied to arrays

Most of the standard mathematical functions can be applied to numpy arrays however you must use the functions defined in the NumPy module.

In [122]:
```
x = array([2, 4, 6, 8])
```

In [123]:
```
cos(x)
```

Out[123]:
```
array([-0.41614684, -0.65364362,  0.96017029, -0.14550003])
```

In [124]:
```
sin(x)
```

Out[124]:
```
array([ 0.90929743, -0.7568025 , -0.2794155 ,  0.98935825])
```

In [125]:
```
log(x)
```

Out[125]:
```
array([ 0.69314718,  1.38629436,  1.79175947,  2.07944154])
```

Plots with Matplotlib

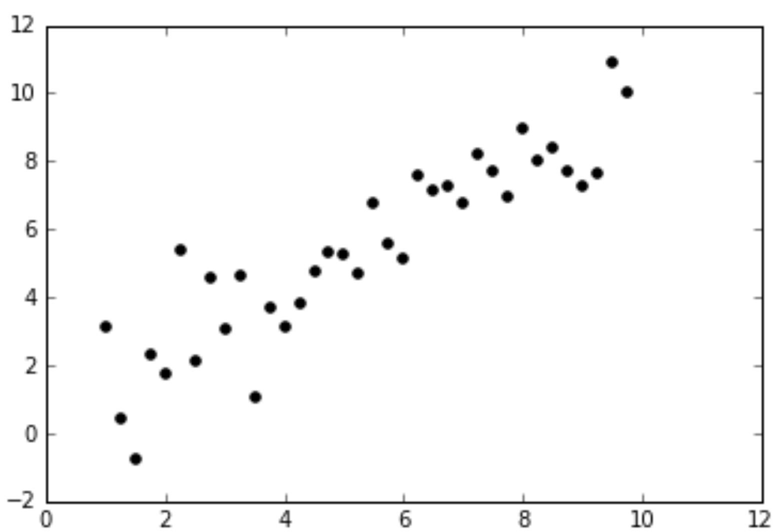
Matplotlib (http://matplotlib.org) is a Python library for making nice 2D and 3D plots. There are a number of other plotting libraries available for Python but matplotlib has probably the most active developer community and is capable of producing publication quality figures.

Matplotlib plots can be generated in a variety of ways but the easiest way to get quick plots is to use the functions defined in the `matplotlib.pyplot` (http://matplotlib.org/api/pyplot_summary.html) module.

In [126]:
```
# this tells Jupyter to draw plots in the notebook itself
%matplotlib inline

# import all the plotting functions from matplotlib.pyplot
from matplotlib.pyplot import *
```

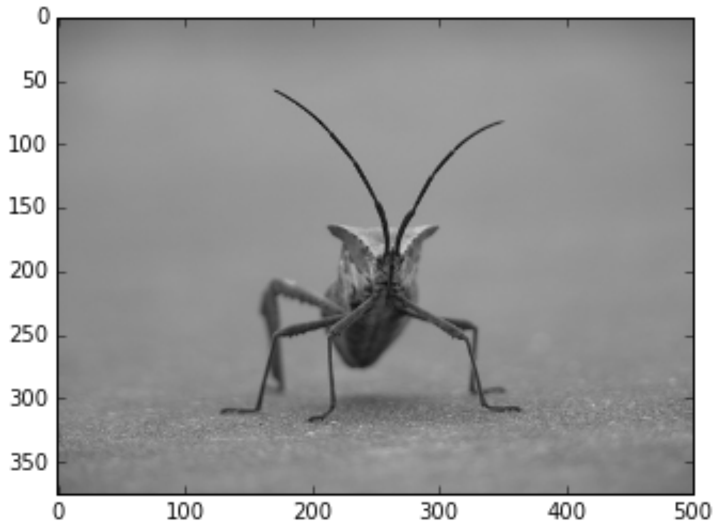
Commonly used functions from `matplotlib.pyplot` include `plot`, `scatter`, `imshow`, `savefig` among others. We explored a decent numbers of plotting functionality at the beginning of this notebook. Here are a few more examples.

In [127]:
```
x = arange(1, 10, 0.25)
y = x + random.normal(size=len(x))
scatter(x,y,color='black')
pass
```

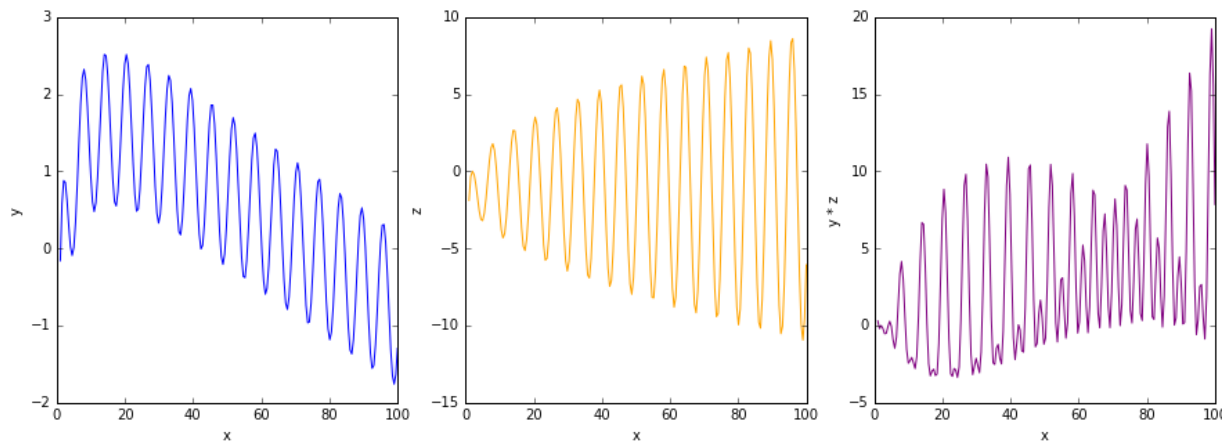


In [128]:

```
# see http://matplotlib.org/users/image_tutorial.html

import matplotlib.image as mpimg # required for loading images
img = mpimg.imread("http://matplotlib.org/_images/stinkbug.png")
imshow(img)

pass
```



In [129]:
```
# demonstrating subplots

fig, (ax1, ax2, ax3) = subplots(nrows=1, ncols=3)
fig.set_size_inches(15,5)

x = linspace(1, 100, 200)
y = log(x**2) - sqrt(x) + sin(x)
ax1.plot(x, y, color='blue')
ax1.set_xlabel("x")
ax1.set_ylabel("y")

z = sqrt(x) * sin(x) - exp(1/x**2)
ax2.plot(x, z, color='orange')
ax2.set_xlabel("x")
ax2.set_ylabel("z")

ax3.plot(x, y*z,color='purple')
ax3.set_xlabel("x")
ax3.set_ylabel("y * z")

pass
```



Strings

Strings aren't numerical data, but working with strings comes up often enough in numerical computing that it's worth mentioning them here.

Strings represent textual information, data or input. String are an interesting data type because they share properties with *data structures* like lists (data structures will be introduced in the next handout).

In [130]:
```
# strings can be enclosed in double quotes
s1 = "Beware the Jabberwock, my son!"
print(s1)
```

```
Beware the Jabberwock, my son!
```

In [131]:
```
type(s1)  # what type are you, s1?
```

Out[131]:
```
str
```

In [132]:
```
# OR in single quotes
s2 = 'The jaws that bite, the claws that catch!'
print(s2)
```

```
The jaws that bite, the claws that catch!
```

In [133]:
```
# If the string you want to write has a quote character
# you need to wrap it in the other type of quote

# note the single quote at the beginning of 'Twas
s3 = "'Twas brillig, and the slithy toves"
print(s3)
```

```
'Twas brillig, and the slithy toves
```

In [134]:
```
# Concatenating (adding) string
s4 = "abc"
s5 = "def"
print(s4 + s5)
```

```
abcdef
```

In [ ]:
```

```