1 Introduction to Python

1.1 About Python

Python is a programming language invented in the early 1980's by a Dutch programmer named Guido van Rossum who was working at the Dutch National Research Institute for Mathematics and Computer Science. Python is a high-level programming language with a simple syntax that is easy to learn. The language supports a variety of programming paradigms including procedural programming, object-oriented programming, as well as some functional programming idioms. The name of the language is a whimsical nod toward Monty Python's Flying Circus.

Python has a very active development community. There is a stable core to the language, but new language features are also being developed. Python has an extensive standard library that includes facilities for a wide range of programming tasks. There is a very large user community the provides support and helps to develop an extensive set of third-party libraries. Python is also highly portable – it is available on pretty much any computing platform you're likely to use. Python is also open-source and free!

1.2 Python Resources

There are many resources available online and in bookstores for learning Python. A few handy resources are listed here:

- Python Website – the official website for the programming language.
- The Python Tutorial – the 'official' Python tutorial.
- Python Library Reference – a reference guide to the many modules that come included with Python.
- Think Python: How to Think Like a Computer Scientist – a free book that provides an introduction to programming using Python.

1.3 Starting the Python interpretter

The Python interpretter can be started in a number of ways. The simplest way is to open a shell (terminal) and type `python`. You can open a terminal as follows:

- On a Mac (OS X) run the terminal program available under `Applications > Utilities`
- On Windows open up a command prompt, available from `Start Menu > Accessories`

Once you're at the command prompt type the following command:

```
python
```

If everything is working correctly you should see something like:

```
Python 2.7.6 |Anaconda 1.7.0 (x86_64)| (default, Nov 11 2013, 10:49:09)
[GCC 4.0.1 (Apple Inc. build 5493)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

If that command didn't work, please see me for further help configuring your Python installation. From within the default interpretter you can type `Ctrl-d` (Unix, OS X) , `Ctrl-z` (Windows) or type `quit()` to stop the interpretter and return to the command line.

1.3.1 IPython

For interactive use, the default interpretter isn't very feature rich, so the Python community has developed a number of GUIs or shell interfaces that provide more functionality. For this class we will be using a shell interface called IPython. `IPython` is one of the tools that was included when you installed the Anaconda Python Distributions.

Again, let's test that things are installed correctly by typing the following command from the terminal:

```
ipython
```

You shoud see the following prompt, which looks like a slightly modified version of the normal Python interpretter:

```
Python 2.7.6 |Anaconda 1.7.0 (x86_64)| (default, Nov 11 2013, 10:49:09)
Type "copyright", "credits" or "license" for more information.

IPython 1.1.0 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object', use 'object??' for extra details.

In [1]:
```

Again, type `quit()` or `Ctrl-D` (Mac) / `Ctrl-Z` (Windows) to exit.

Recent versions of IPython provides both terminal and GUI-based shells. For a GUI based version of the ipython intepretter type the following at the command prompt.

```
ipython qtconsole
```

QtConsole is a recent addition to IPython and there may still be bugs to be sorted out, but it provides some very nice features like 'tooltips' (shows you useful information about functions as you type) and the ability to embed figures and plots directly into the console, and the ability to save a console session as a web page (with figures embedded!).

For most of the this class we'll be using several numerical (Numpy, Scipy) and plotting (Matplotlib) libraries that are frequently used for scientific computing in Python. We can import these libraries automatically by appending the argument `--pylab` to the ipython command, as shown below:

```
ipython --pylab
```

To get the equivalent of `QtConsole` you can run ipython with the following arguments:

```
ipython qtconsole --pylab=inline
```

Here we added an option to the `--pylab` argument that will cause figures to be printed inline in the GUI window. To see how this works, type the following line into the ipython qtconsole window:

```
In [1]: plot(range(10))     # don't type the bit that says "In [1]:"
```

1.3.2 Quick IPython tips

IPython has a wealth of features, many of which are detailed in its documentation. There are also a number of videos available on the IPython page which demonstrate some of it's power. Here are a few key features to get you started and save you time:

- *Don't retype that long command!* — You can scroll back and forth through your previous inputs using the up and down arrow keys (or `Ctrl-p` and `Ctrl-n`); once you find what you were looking forward you can edit or change it. For even faster searching, start to type the beginning of the input and then hit the up arrow.

- *Navigate using standard Unix commands* — IPython lets you use standard Unix commands like `ls` and `cd` and `pwd` to navigate around your file system (even on Windows!)

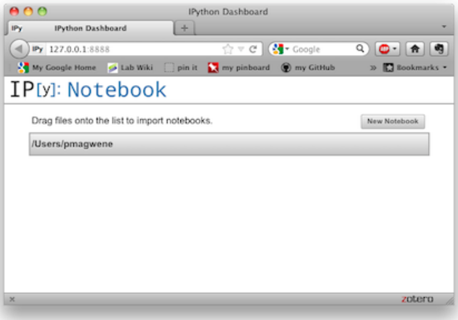
- *Use <Tab> for command completion* — when your navigating paths or typing function names in you can hit the `<Tab>` key and IPython will show you matching functions or filenames (depending on context). For example, type `cd ./<Tab>` and IPython will show you all the files and subdirectories of your current working directory. Type a few of the letters of the names of one of the subdirectories and hit `<Tab>` again and IPython will complete the name if it finds a unique match. Tab completeion allows you to very quickly navigate around the file system or enter function names so get the hang of using it.

1.3.3 IP[y] Notebooks

For most of this class we'll be using a web-browser based 'notebook' to interface with Python. This notebook tool, called `IP[y]`, is included with `IPython`. `IP[y]` notebooks are similar to Mathematica notebooks, in that you can weave together explanatory text and code. To start an `IP[y]` notebook first open up a terminal or command prompt and type the following command:

```
ipython notebook --pylab=inline    # note the use of the --pylab=inline
    argument again!
```

If `IPython` was installed correctly this will open up a new tab or window in your webbrowser, as show in Fig. 1.1. Click the "New Notebook" button in the upper right and you'll be presented with an interface like the one show in Fig. 1.2.



Drag files onto the list to import notebooks.

/Users/pmagwene

Figure 1.1: The web-browser based `IP[y]` Notebook is a new feature of `IPython`, available in version 0.12.

1.3.4 Entering commands in IP[y] Notebooks

Unlike the normal Python interpreter, when you hit `Enter` in an `IP[y]` notebook, the commands you enter in a notebook cell are not immediately evaluated. You have to use `Shift-Enter` (hold the `Shift` key while you hit the `Enter` key) when you want a cell to be evaluated.

In the examples that follow, lines that begin with `In` indicate lines that you should enter in the `IP[y]` notebook. You should not type the `In:` part of these commands, this merely show you the prompt in the interpretter. Lines that follow will indicate the output produced by that command (sometimes the output will be omitted).

1.3.5 Accessing the Documentation

Python comes with extensive HTML documentation. If you have a network connection, you can access the online documentation for Python (and several other packages) by

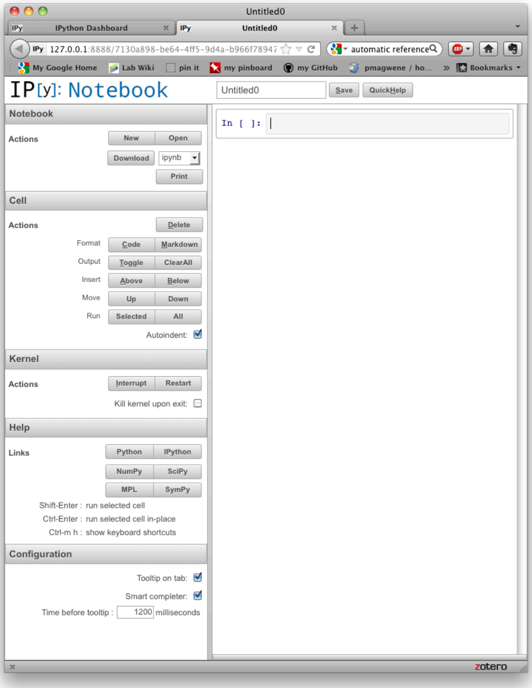


Figure 1.2: The IP[y] Notebook interface.

clicking the appropriate button under "Help" in the left-hand frame of IP[y]. Alternately, you can use the help() function (a built-in function in Python), or the ? command (specific to IPython):

```
In: help(len)
In: ?len
```

1.4 Using Python as a Calculator

Alert!

When entering these lines from inside IP[y], remember to hit Shift-Enter to evaluate the commands!

The simplest way to use Python is as a fancy calculator. Let's explore some simple arithmetic operations:

```
In: 2 + 4 # addition
Out: 6
```

```
In: 2 - 4 # subtraction
Out: -2

In: 2 * 4 # multiplication
Out: 8

In: 2/4 # division
Out: 0   # huh, what's going on here!

In: 2.0/4.0  # floating point division
Out: 0.5     # that's more along the lines of what you might expect!

In: (10+2)/(4-5)
Out: -12

In: (10+3)/4-5 # compare this answer to the one above
Out: -2

In: 2**4  # exponentiation , i.e. 2 raised to the power 4
Out: 16
```

As you can see from the division example above, Python distinguishes between integer values and real numbers. So when we tried `2/4` the corresponding output couldn't be represented as an integer so Python rounded the result to the nearest integer value. By contrast, when we used floating point values, `2.0/4.0`, Python returned the answer we expected.

Numerical Data Types

Python recognizes three built-in types of numerical values: integers, floating point values (real numbers), and complex numbers.

```
In: 3 # an integer
Out: 3

In: type(3)
Out: int

In: 3.0 # floating point
Out: 3.0

In: type(3.0)
Out: float

In: 3 + 0j
Out: (3+0j)

In: type(3 + 0j) # complex number; engineers use 'j' to represent imaginary
     numbers
```

```
Out: complex

In: (1 + 2j) + (0 + 3j)  # adding complex numbers
Out: (1+5j)

In: (3 + 1j) * (4 + 2j)  # complex multiplication
Out: (10+10j)
```

When performing arithmetic operations using a mixture of types, the returned values are of the more ‘general’ type.

```
In: x = 2
In: y = 2.0
In: z = 2 + 0j

In: x
Out: 2

In: x + y
Out: 4.0

In: x + y + z
Out: (6+0j)
```

Some things to remember about mathematical operations in Python:

- Integer and floating point division are not the same in Python. Generally you’ll want to use floating point numbers.
- Be aware that certain operators have precedence over others. For example multiplication and division have higher precedence than addition and subtraction. Use parentheses to disambiguate potentially confusing statements.

Standard math functions like `cos()` and `log()` are not available to the Python interpeter by default. To use these functions you usually need to `import` the math library as shown below. This doesn’t apply when you’ve started Ipython using the `--pylab` argument, which automatically imports the commonly used math functions. However, it’s still important that you understand what’s “built-in” versus imported from other libraries.

```
In: import math  # make the math module available
In: dir(math)    # show the variables and fxns defined in the math module
Out: ['__doc__', '__file__', '__name__', '__package__', 'acos', 'acosh', '
    asin', ...
... etc ... ]

In: math.cos(0.5)
Out: 0.8775825618903728

In: math.pi
Out: 3.141592653589793
```

```
In: math.cos(math.pi)
Out: -1.0

# this works because the --pylab argument automatically imported common
    functions
# and variable names
In: cos(pi)
Out: -1.0
```

1.4.1 Comparison Operators in Python

Comparison operators in Python allow you to test the truthfullness of various statements or conditions. They typically return Boolean values (see below). The basic comparison are shown below:

```
In: 10 < 9 # less than
Out: False

In: 10 > 9 # greater than
Out: True

In: 10 <= (5 * 2) # less than or equal to
Out: True

In: 10 >= pi  # greater than or equal to
Out: True

In: 10 == 10  # test equality
Out: True

In: 10 != 10  # test inequality
Out: False

In: 10 == (sqrt(10)**2)  # Surprised by this result? See below.
Out: False

In: 4 == (sqrt(4)**2) # Even more confused?
Out: True
```

Be careful to distinguish between == (tests equality) and = (assignment operator).

How about the last two statement comparing two values to the square of their square roots? Mathematically we know that both $(\sqrt{10})^2 = 10$ and $(\sqrt{4})^2 = 4$ are true statements. Why does Python tell us the first statement is false? What we're running into here are the limits of computer precision. A computer can't represent $\sqrt{10}$ exactly, whereas $\sqrt{4}$ can be exactly represented. Precision in numerical computing is a complex subject and beyond the scope of this course.

1.5 Boolean and String Data Types

You've already seen the three basic numeric data types in Python - integers, floating point numbers, and complex numbers. There are two other basic data types - Booleans and strings.

1.5.1 Boolean values

Here's some examples of using the Boolean data type:

```
In: x = True
In: type(x)
Out: <type 'bool'>

In: y = False
In: x == y
Out: False

# don't input the '...' below, this simply shows the continuation of the
    input
In: if x is True:
...     print 'Oh Yeah!'
...
Oh Yeah!

In: if y is True:
...     print 'You betcha!'
... else:
...     print 'Sorry, Charlie'
...
Sorry, Charlie
```

1.5.2 Strings

Here's some examples of using the string data type:

```
In: s1 = 'It was the best of times'
In: type(s1)
Out: <type 'str'>

In: s1 # show representation of string
Out: 'It was the best of times'

In: print s1 # print the string, notice difference from above
It was the best of times

In: s2 = 'it was the worst of times'
In: s1 + s2   # string concatenation
```

```
Out: 'It was the best of timesit was the worst of times'

In: s1 + ', ' + s2
Out: 'It was the best of times, it was the worst of times'

In: 'times' in s1   # test whether 'times' is a substring of s1
Out: True

In: s1.count('t')  # count the occurences of 't' in s1
Out: 4

In: s3 = "You can nest 'single quotes' in double quotes"
In: s4 = 'or "double quotes" in single quotes'
In: s5 = "but you can't nest "double quotes" in double quotes"
  File "<ipython-input-38-1d7699ed8482>", line 1
    s4 = "but you can't nest "double quotes" in double quotes"
                                   ^
SyntaxError: invalid syntax
```

Note that you can use either single or double quotes to specify strings.

1.6 Elementary data structures in Python

1.6.1 Python Lists

Lists are the simplest 'built-in' data structure in Python. List represent ordered collections of arbitrary objects.

```
In: l = [2, 4, 6, 8, 'fred']
In: l
Out: [2, 4, 6, 8, 'fred']

In: len(l)  # query list for its length
Out: 5
```

Python lists are zero-indexed. This means you can access lists elements in the range `0` to `len(x)-1`.

```
In: l[0]
Out: 2

In: l[3]
Out: 8

In: l[5]
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-10-437134752604> in <module>()
----> 1 l[5]
```

```
IndexError: list index out of range
```

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

```
In: l[-1]  # the last element
Out: 'fred'

In: l[-2]  # the 2nd to last element
Out: 8

In: l[-3]  # ... etc ...
Out: 6
```

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

```
In: l = [2, 4, 6, 8, 'hike!']
In: l[-1]
Out: 'hike!'

In: l[-1] = "learning python is so great!"
In: l
Out: [2, 4, 6, 8, 'learning python is so great!']
```

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

```
In: l1 = [1,2,3]
In: l2 = ['a', 'b', 'c', 'd']
In: l1.append(4)
In: l1
Out: [1, 2, 3, 4]

In: del l2[2]
In: l2
Out: ['a', 'b', 'd']

In: l3 = l1 + l2
In: l3
Out: [1, 2, 3, 4, 'a', 'b', 'd']
```

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

```
In: y = range(10)  # use help(range) to read about this fxn
In: y
Out: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

In: y[:3]  # get first 3 elements of y
Out: [0, 1, 2]

In: y[2:8] # get elements from 2 to 8
Out: [2, 3, 4, 5, 6, 7]

In: y[2:-1] # get elements from 2 to the last element
```

```
Out: [2, 3, 4, 5, 6, 7, 8]

In: y[-1:0] # how come this didn't work?
Out: []

# slice from last to first, stepping backwards by 2
In: y[-1:0:-2]
Out: [9, 7, 5, 3, 1]
```

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

```
In: s = ['a', 'b', 'c', 'd', 'e']
In: s[2:4] = ['C', 'D']
Out: ['a', 'b', 'C', 'D', 'e']
```

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

```
In: l4 = [1, 5, 3, 4, 10, 11, 3]
In: l4.sort() # sort in-place
In: l4
Out: [1, 3, 3, 4, 5, 10, 11]

In: l4.reverse() # reverse in-place
In: l4
Out: [11, 10, 5, 4, 3, 3, 1]
```

Read, 'Think Python' Chapter 10 for more about lists. See the Python library documentation for a summary of methods associated with lists.

1.6.2 Python Tuples

As shown above, the elements of a Python list are mutable and the length of lists can be changed. Sometimes it's useful for reasons of both programming logic and computational efficiency to have immutable collections of items. The data structure that Python uses to represent such immutable collection is called a 'Tuple'.

Parentheses rather than square brackets are used to create tuples. A common use of tuples is to represent a pair, i.e. a tuple of length two. For example, you might use a pair to represent a point in the Cartesian plane:

```
In: pt = (1,5)
In: pt
Out: (1, 5)

In: pt[0]
Out: 1

In: pt[1]
Out: 5

In: pt[0] = 2
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-50-648d2cfb62f4> in <module>()
----> 1 pt[0] = 2

TypeError: 'tuple' object does not support item assignment
```

The above illustrates that tuples can be indexed like lists, however you can't assign to or extend a tuple after it's created. As we will illustrate later, tuples are often used to return multiple objects from functions. Read, 'Think Python' Chapter 12 for more about tuples.

1.6.3 Python Dictionaries

Python dictionaries (sometimes called 'hash maps' in other programming languages) map a set of keys (labels) to values. Dictionaries provide efficient access to non-ordered data.

```
## create a dictionary
In: d = {'fred':26, 'jim':22, 'mary':34, 'jill': 12}

In: d
Out: {'jim': 22, 'mary': 34, 'jill': 12, 'fred': 26}

## get value associated with the key 'fred'
In: d['fred']
Out: 26

## return list of keys in dict
In: d.keys()
Out: ['jim', 'mary', 'jill', 'fred']

## return list of values in dict
In: d.values()
Out: [22, 34, 12, 26]

## iterate over the keys in the dict
In: for key in d.keys():
...     print key, "is", d[key], "years old."
...
jim is 22 years old.
mary is 34 years old.
jill is 12 years old.
fred is 26 years old.

## add a new key,value pair
In: d['joe'] = 99
In: d
Out: {'joe': 99, 'jim': 22, 'mary': 34, 'jill': 12, 'fred': 26}
```

```
## delete a key and it's associated value
In: del d['mary']
In: d
Out: {'joe': 99, 'jim': 22, 'jill': 12, 'fred': 26}
```

Dictionary keys have to be immutable objects, but associated values can be arbitrary python objects, even other dictionaries.

```
## create an empty dictionary
In: d2 = {}

## a list can't be a dictionary key because it's mutable
In: d2[[1,2,3]] = 'b'
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-62-497f3cfcc4ef> in <module>()
----> 1 d2[[1,2,3]] = 'b'

TypeError: unhashable type: 'list'

## but a tuple can
In: d2[(1,2,3)] = 'b'
In: d2
Out: {(1, 2, 3): 'b'}

## an alternate way to create an empty dict
In: d3 = dict()

# create a dictionary of dictionaries
In: d3['fred'] = {'age': 26, 'sex': 'male', 'occupation':'nurse'}
In: d3['mary'] = {'age': 34, 'sex': 'female', 'occupation':'engineer'}

In: d3['fred']['age']
Out: 26

In: d3['mary']['occupation']
Out: 'engineer'
```

Read, 'Think Python' Chapter 11 for more about dictionaries.

1.7 Control Structures in Python

Control structures allow you to control the flow of execution in a program. For example, you can use a for loop to carry out a calculation on every object in a list.

```
In: x = [0, 2, 4, 6, 8]
In: for item in x:  # iterates over every item in the list x
...     print item, item**2
```

```
...
0 0
2 4
4 16
6 36
8 64
```

A programming idiom related to a `for` loop is what is called a 'list comprehension.' List comprehensions come from functional programming languages like Haskell. List comprehensions provide a concise way to create lists by applying a function to another list. The following code illustrates the use of a list comprehension:

```
In: x = [0, 2, 4, 6, 8]
In: x2 = [item**2 for item in x]
In: print x2
[0, 4, 16, 36, 64]
```

You can use `if...else` blocks to execute certain commands only if a particular statement is true:

```
In: x = [1,2,3,4,5,6,7,8]
In: for item in x:
...     if item % 2:    # tests for odd numbers
...             print item, 'is odd'
...     else:
...             print item, 'is even'
...
1 is odd
2 is even
3 is odd
4 is even
5 is odd
6 is even
7 is odd
8 is even
```

Another common control structure is a `while` loop:

```
In: x = [0, 2, 4, 6, 8]
In: i = 0
In: while x[i]**2 < 50:
...     print x[i], x[i]**2
...     i += 1  # same as writing i = i + 1
...
0 0
2 4
4 16
6 36
```

1.7.1 Using 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/. NumPy does not come with the standard Python distribution, but it does come as an included package if you use the Enthought Python distribution. The NumPy package comes with documentation and a tutorial. You can access the documentation here: 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).

```
In: x = array([2,4,6,8,10]) # the array object is defined in numpy
In: type(x)
<type 'numpy.ndarray'>

In: x.dtype # the type of elements in x
dtype('int32')

In: -x
array([ -2,  -4,  -6,  -8, -10])

In: x**2
array([  4,  16,  36,  64, 100])

In: a = x * pi # assumes pi is in the current namespace
In: a
array([  6.28318531,  12.56637061,  18.84955592,  25.13274123,
    31.41592654])

In: type(a)
<type 'numpy.ndarray'>

In: a.dtype
dtype('float64')

In: y = array([0, 1, 3, 5, 9])
In: x + y
array([ 2,  5,  9, 13, 19])

In: x * y
array([ 0,  4, 18, 40, 90])

In: z = array([1, 4, 7, 11])
In: x + z
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: shape mismatch: objects cannot be broadcast to a single shape

In: len(x)
5

In: len(z)
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: m = array([[1,2],
...            [3,4]])
In: m
array([[1, 2],
       [3, 4]])

In: m.transpose()
array([[1, 3],
       [2, 4]])

In: from numpy import linalg
In: linalg.inv(m)
array([[-2. ,  1. ],
       [ 1.5, -0.5]])
```

Division by zero for normal Python floats produce a `ZeroDivisionError` but division by zero for numpy arrays returns a result indicating an infinite number (`Inf}`:

```
In: 10/0.0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: float division
In: x
array([ 2,  4,  6,  8, 10])
In: x/0.0
array([ Inf,  Inf,  Inf,  Inf,  Inf])
```

1.7.2 Indexing and Slicing NumPy Arrays

Like the built-in lists, arrays in NumPy are zero-indexed rather than one-indexed:

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

```
In: len(x)
5

In: x[0]
2

In: x[1]
4

In: x[4]
10

In: x[5]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: index out of bounds
```

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

```
In: x[-1] # the last element
10

In: x[-2] # the 2nd to last element
8

In: x[-3] # ... etc ...
6

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

In: y = array(range(15))
In: y
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14])

In: y[2:14:3] # give me elements 2 to 14 of y, with a stride of 3
array([ 2,  5,  8, 11])
```

1.7.3 Comparison operators for arrays

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

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

In: x < 5
array([ True,  True, False, False, False], dtype=bool)

In: x >= 6
```

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

1.7.4 Combining Indexing and Comparison

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

```
In: x = array([2,4,6,8,10])
In: x[x > 5]
array([ 6,  8, 10])

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

In: x[logical_or(x < 4, x > 6)]
array([ 2,  8, 10])
```

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 at http://docs.scipy.org/doc/numpy/reference/routines.logic.html for more info on numpy logical functions.

1.7.5 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: range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

In: range(3,12,4) # generate numbers from 3 to 12 (non-inclusive) stepping
    by 4
[3, 7, 11]

In: range(1,10,0.5)  # doesn't work!
__main__:1: DeprecationWarning: integer argument expected, got float
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: range() step argument must not be zero

In: numpy.arange(1,10,0.5)  # note the use of numpy `arange' function
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: [True,True,False]*3
[True, True, False, True, True, False, True, True, False]
```

1.7.6 Mathematical functions applied to NumPy 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: x
array([ 2,  4,  6,  8, 10])

In: import math
In: math.cos(x)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: only length-1 arrays can be converted to Python scalars

In: import numpy
In: numpy.cos(x)
array([-0.41614684, -0.65364362,  0.96017029, -0.14550003, -0.83907153])
```

1.7.7 More Numerical Functions

You've already seem a number of functions (e.g. `sin()`, `log()`, `len()`, etc). Functions are called by invoking the function name followed by parentheses containing zero or more *arguments* to the function. Arguments can include the data the function operates on as well as settings for function parameter values. We'll discuss function arguments in greater detail below.

There are a number of statistical functions built into NumPy (and even more in `SciPy`). Here we'll illustrate some of the more common ones:

```
In: test_scores = [98, 92, 78, 65, 52, 59, 75, 77, 84, 31, 83, 72, 59, 69,
    71, 66]
In: min(test_scores)
31

In: max(test_scores)
98

In: numpy.mean(test_scores)
70.6875

In: numpy.median(test_scores) # the median is another measure of centrality
71.5
```

```
In: numpy.var(test_scores, ddof=1) # variance is a measure of the spread of
     a distn
259.82916666666665

In: numpy.std(test_scores, ddof=1) # standard deviation = sqrt(var)
16.119217309369169
```

In the code above we use the argument ddof=1 to indicate that we want the so-called 'unbiased' versions of these statistics (most appropriate for small samples).

1.8 Writing Your Own Functions

You've now explored some of Python's built in functions as well as those included in the math module and the numpy module. Now let's take a look at writing your own functions.

The general form of a Python function is as follows:

```
def funcname(arg1, arg2):
    # one or more expressions
    return someresult # arbitrary python object (could even be another
        function)
```

An important thing to remember when writing functions is that Python is white space sensitive. In Python code indentation indicates scoping rather than braces. Therefore you need to maintain consistent indendation. This may surprise those of you who have extensive programming experience in another language. However, white space sensitivity contributes signficantly to the readability of Python code. Use a Python aware programmer's editor and it will become second nature to you after a short while.

Here's an example of defining and using a function in the Python interpreter:

```
In: import math
In: def sinsqrd(x):
  '''A function for calculating sine squared.'''
  return math.sin(x)**2

In: sinsqrd(math.pi/4)
0.49999999999999989

In: import numpy
In: x = numpy.arange(1,20,0.1)
In: siny = [math.sin(item) for item in x]  # using list comprehensions
    again!
In: sinsqrdy = [sinsqrd(item) for item in x]
```

The sinsqrd function defined above only works on integers or floats (that is why we used the list comprehension to calculate the values over the range of x). If you wanted to write an equivalent function that also works on lists and arrays you could write it like this:

```
In: def sinsqrd2(x):
...     return numpy.sin(x)**2
...
In: ysin2 = sinsqrd2(x)
```

1.9 Making Plots with Matplotlib

matplotlib 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 pylab functions that matplotlib provides. The pylab function are meant to be very similar to the plotting function in Matlab. As with NumPy, when starting IPython with the --pylab argument all the functions defined in pylab are imported and made available to you.

```
# the array() function was made available when we started the interpetter
# with the --pylab option
# array() is part of the Numpy library
In: x = array([1,2,3,4,5,6,7,8,9,10])

## the plot() function comes from the Matplotlib library
In: plot(x, x**2)
```

Now click on the notebook cell with the plot command, change it to the following, and hit Shift-Enter to re-evaluate the cell.

```
plot(x, x**2, color='red', marker='o')
xlabel("Length")
ylabel("Area")
title("Length vs. Area for Squares")
```

One of the coolest features of IP[y] notebooks is that they allow you to interactively enter some code, evaluate the results, and then go back and fix, edit or change the code and re-evaluate it without having to reload or compile anything. This is particularly useful for interactively creating complex graphics.

Here we show how to generate a figure that illustrates the sin function and the sinsqrd function we defined above:

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

# the color ('r-') and label arguments used below
plot(x, sinx, 'r-', label='sin(x)')
plot(x, sinsqrx, 'b-', label='sin^2(x)')
legend(loc='best') # add an optional legend
```

1.9.1 More Matplotlib examples

Let's create a more complicated figure, illustrating a histogram of random draws from a normal distribution, compared to the expected probability distribution function (PDF) for a normal distribution with the same parameters:

```
mean = 100
sd = 15

# draw 1000 random samples from a normal distn
normaldraw = normal(mean, sd, size=1000)

# draw a histogram
# "normed" means normalize the counts
n, bins, patches = hist(normaldraw, bins=50, normed=True)
xlabel("x")
ylabel("density")

# draw the normal PDF for the same parameters
# evaluated at the bins we used to construct the histogram
y = normpdf(bins, mean, sd)
l = plot(bins, y, "r--", linewidth=2)
```

This produces the plot shown in Fig. 1.3.

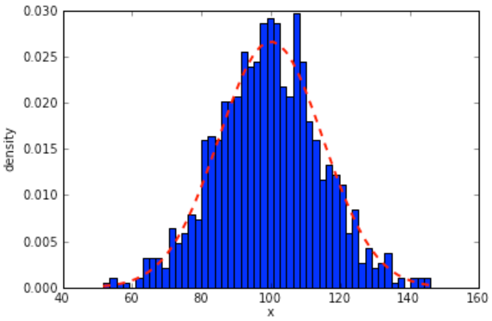


Figure 1.3: A histrogram created using `IP[y]` and the `matplotlib` library.

In this final set of examples we create several representations of the function $z = cos(x)\,sin(y)$.

```
def f(x,y):
    # multiply by pi/180 to convert degrees to radians
    return cos(x*pi/180) * sin(y*pi/180)

# note we set the upper boundary as 361
# so that 360 get's included
x,y = ogrid[0:361:10, 0:361:10]
z = f(x,y)
```

```
# ravel insures the x and y are 1d arrays
# try with 'contour' rather than 'contourf'
p = contourf(ravel(x), ravel(y),z)

lx = xlabel("x (degrees)")
tx = xticks(arange(0,361,45))
ly = ylabel("y (degrees)")
tx = yticks(arange(0,361,45))
```

And the same function represented in 3D, that produces Fig. 1.4.

```
from mpl_toolkits.mplot3d import Axes3D
fig = figure()
ax = Axes3D(fig)

# create x,y grid
x,y = meshgrid(arange(0,361,10), arange(0,361,10))
z = f(x,y) # uses fxn f from previous cell
ax.plot_surface(x,y,z,rstride=2,cstride=2,cmap="jet")

# setup axes 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)
fig.show()
```

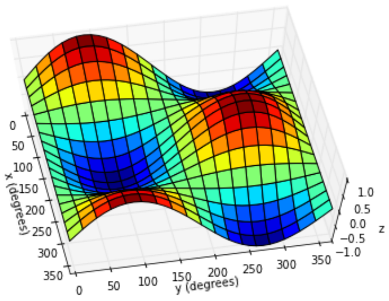


Figure 1.4: A 3D representation of $z = cos(x)\, sin(y)$