Doing Science with Python

v.0.4.1

Jesus Perez Colino

March 2015

0.1 Something about the author...

Documention prepared by Jesus Perez Colino.Version 0.4.1, Released 01/04/2015, Beta

-
- Comments and suggestions are always welcome. You can contact me by email: `jpcolino@gmail.com`
- **Acknowledge**: To create this document I have inserted many examples from many people and many places. Specially thanks to **Edward Tufte** from Stanford, **Hanspeter Pfister**, **Joe Blitzstein** and **Chris Beaumont** from Harvard University for some of the examples and, of course, big thanks to the people who develop **NumPy**, **SciPy**, **IPython**, **pandas**, **Scikit-learn**, **Statsmodels** and **CVXOpt** to leave it for everybody for free.
- This work is offered for free, with the hope that it will be useful. Enjoy!

CHAPTER

ONE

SOME BASICS BEFORE TO START...

“If builders built buildings the way programmers wrote programs,
then the first woodpecker that came along would destroy civilization“
– Gerald M. Weinberg, "The Psychology of Computer Programming", 1971

1.1 Why Python for Data Analysis?

- Python is great for scripting and applications.
- The `pandas` library offers improved Time Series and Data Framework support.
- Easy Web Scraping, web APIs
- Strong High Performance Computation support
  - Load balanceing tasks
  - MPI, GPU
  - MapReduce
- Strong support for abstraction
  - Intel MKL
  - HDF5
- IPhyton (Notebook) for presentation and communication of ‘reproducible’ results

1.2 Why not Python?

- R, MATLAB are great… and C++ is still the King. Julia is promising.
- Slow compared with C++ or Fortran
- Visualization
  - Matplotlib is 10 years old now
  - Rob Story’s vicent
  - Bokeh
  - R’s ggplot2 is amazing

1.3 References

There is a huge amount of info and examples in Internet. I didn’t come up with all the examples!

- Python Tutorial
- Learn Python the Hard Way
- IPython notebook documentation or the IPython tutorial

- Python for Data Analysis
- J.R. Johansson's Lectures in Scientific Computing, excellent notebooks, including Scientific Computing with Python and Computational Quantum Physics with QuTiP lectures;
- XKCD style graphs in matplotlib;
- A collection of Notebooks for using IPython effectively
- A gallery of interesting IPython Notebooks

1.4 What You Need to Install

To do this course, you need a Python package installed that should includes:

- Python version 2.7.9;
- NumPy, the core numerical extensions for linear algebra and multidimensional arrays;
- SciPy, additional libraries for scientific programming;
- Matplotlib, excellent plotting and graphing libraries;
- Pandas, easy-to-use data structures and data analysis tools;
- Scikit-Learn, for statistical modelling and machine learning;
- IPython, with the additional libraries required for the notebook interface.

An easy to install for Mac, Windows, and Linux, with all these packages is the Anaconda Python Distribution, from Continuum Analytics

Alternatives:

- **Linux:** Most distributions have an installation manager. Redhat has yum, Ubuntu has apt-get.
- **Mac:** Macports
- **Windows:** Anaconda, PythonXY or Canopy
- **Cloud:** Check out Wakari, from Continuum Analytics, which is a cloud-based IPython notebook.

Let us check, first, if every package is correctly installed in your computer. Go to the **next cell**, click on with the mouse and press **Shift-Intro** to execute the next cell.

1.5 Reproducibility Conditions

```
In [1]: %matplotlib inline
        import os
        import math
        import ipython
        import time as t
        import numpy as np
        import pandas as pd
        from sys import version
        from scipy import stats
        import matplotlib.pyplot as plt
        import matplotlib as mpl
        #os.chdir('C:\\Users\\Suso')
        print '='*100
        print 'Python version:     ' + version
        print 'NumPy version:      ' + np.__version__
        print 'Pandas version:     ' + pd.__version__
        print 'Matplotlib ver:     ' + mpl.__version__
        print 'IPython version:    ' + IPython.__version__
        direct = %pwd
        print 'Working directory:  ' + direct
        print '='*100
        now = t.asctime()
        print 'Today is ' + now + ' ... AND WE ARE READY TO GO!! '
        print '='*100
```

```
===============================================================================================
Python version:     2.7.9 |Anaconda 2.2.0 (x86_64)| (default, Dec 15 2014, 10:37:34)
[GCC 4.2.1 (Apple Inc. build 5577)]
NumPy version:      1.9.2
Pandas version:     0.15.2
Matplotlib ver:     1.4.3
IPython version:    3.0.0
Working directory:  /Users/JPC/Dropbox/IPython Notebooks
===============================================================================================
Today is Mon Apr  6 13:39:45 2015 ... AND WE ARE READY TO GO!!
===============================================================================================
```

CHAPTER

TWO

GETTING STARTED WITH PYTHON

“Let us change our traditional attitude to the construction of programs:
Instead of imagining that our main task is to instruct a computer what to do,
let us concentrate rather on explaining to humans what we want the computer to do.“
– Donald E. Knuth, "Literate Programming", 1984

2.1 What is Python?

Python is a general-purpose, high-level and multi-paradigm programming language.

Its design philosophy emphasizes code readability, and its syntax allows programmers to express concepts in fewer lines of code than would be possible in languages such as C++.

- Simple, clean syntax
- Easy to learn
- Interpreted or Compiled
- Dynamically typed
- Multi-platform: Linux, Mac, and Windows
- Free, as in beer and speach
- Expressive: do more with fewer lines of code
- Lean: **modules**

2.2 History

- Created by Dutch programmer **Guido van Rossum**, 1989, declared *Benevolent Dictator for Life (BDFL)* by Python community.
- Conceived as a successor to ABC language and interfacing with Amoeba distributed OS.
- Initially the syntax came from `C`, but with less exceptions and special cases
- **Multi-paradigm**: accept object-oriented, structured and functional programming
- It brings from `LISP` (functional prog.) the `lambda`, `reduce`, `filter`, and `map`
- `Modula-3` inspired keyword arguments and `imports`
- **`Python 2.0`** was released on Oct-2000, with many major new features including a full garbage collector and support for Unicode.
- **`Python 3.0`** was a backwards-incompatible version released on Dec-2008. Many of its major features have been backported to the backwards-compatible `Python 2.6` and `Python 2.7`

2.3 What can you do with Python?

- Scripting language
  - OS support
  - glue existing applications
- Scientific abstraction
  - Use highly optimized C and Fortran libraries
  - Intel MKL, HDF5, Blas, Lapack, MPI, Cuda
- Data Analysis
- Visualization
- Scrape websites
- Build websites
- Anything!

2.4 Performance

- Minimal time required to develop code
- Rapid prototyping
- Interpreted and dynamically typed means it's slower
  - Use optimized libraries: e.g. NumPy , SciPy
  - Find the bottleneck and write it in C/C++/Fortran: e.g. f2py, cython
  - Just-in-time: e.g. numba
- Implementing the built-in Python modules would require some advanced programming skills in other languages

2.5 Advantages

- Interpretive
- Named function arguments
- Heterogeneous data structures
  - dictionary
  - lists
- No memory management
- Print structures
- Packages: large community of support
- Modular: great for larger projects (modules)

2.6 Modules for Data Analysis

Base:

- IPython (renamed as jupyter): ipython or jupyter is the component in the standard scientific Python toolset that ties everything together. It provides a robust and productive environment for interactive and exploratory computing. It is an enhanced Python shell designed to accelerate the writing, testing, and debugging of Python code. It is particularly useful for interactively working with data and visualizing data with matplotlib. Some of the most useful features are:
  - Tab completion
  - History mechanism

  - Inline editing
  - Ability to call external Python scripts with %run
  - Access to system commands
  - The pylab switch
  - Access to Python debugger and profiler
- NumPy: NumPy provides a multidimensional array object to store homogenous or heterogeneous data; it also provides optimized functions/methods to operate on this array object.
- SciPy: SciPy is a collection of libraries and functions implementing important standard functionality often needed in science or finance; for example, you will find functions for cubic splines interpolation as well as for numerical integration.
- Matplotlib: This is the most popular plotting and visualization library for Python, providing both 2D and 3D visualization capabilities.
- Pandas: pandas builds on NumPy and provides richer classes for the management and analysis of time series and tabular data; it is tightly integrated with matplotlibNumPy for plotting and PyTables for data storage and retrieval.
- Scikit-Learn: statistical modelling and machine learning

Additional:

- Requests: data from the web
- BeautifulSoup: parse web pages
- Pytables: fast, binary files
- Sqlite3: binding for sqlite3
- Spark: in-memory MapReduce
- Disco: MapReduce

CHAPTER

THREE

PYTHON FIRST STEPS

3.1 Using Python as Calculator

The easiest way to begin to work with **Python** is holding a direct conversation with the **Python interpreter**, where you can speak in expressions and it replies with evaluations. There is clearly a **read - eval - print** loop going on under the hood. Let us begin with the simplest possible mathematical expressions: as an **on-line calculator** (If you're typing this into an IPython notebook, press **Shift-Enter** to evaluate a cell.)

```
In [2]: 2+2*3098098

Out[2]: 6196198

In [3]: 7/3

Out[3]: 2

In [4]: 7/3.

Out[4]: 2.3333333333333335
```

Python provides a full set of arithmetic operators, including binary operators for the four basic mathematical operations: + addition, − subtraction, * multiplication, and / division. In addition, many Python data types can be used with augmented assignment operators such as += and *=.

Related with the data type of the inputs, let us start with the most basic data types: **integers**, or *whole numbers* to the non-programming world, and **floating-point numbers** or float type, also known (incorrectly) as *decimal numbers* to the rest of the world.

Python integer division, like C or Fortran, truncates the remainder and returns an integer. In Python 3, returns a floating point number, by default.

```
In [5]: print type(7/3)

<type 'int'>


In [6]: print type(7/3.)

<type 'float'>


In [7]: 7/3 == 7/float(3)

Out[7]: False
```

The **float** or **floating-point type** supports the normal arithmetic operations, as well as several rounding options. Python floats are based on the Institute of Electrical & Electronic Engineering (IEEE) standards and have the same ranges as the underlying computer architecture. They also suffer the same levels of imprecision that make comparing float values a risky option (see next section *'Some Pitfalls in Numerical Computing'*).

Python provides three kinds of floating-point values: the built-in **float** and **complex** types, and the **decimal.Decimal** type from the standard library. All three are *immutable*. Type float holds double-precision floating-point numbers whose range depends on the C (or C or Java) compiler Python was built with; they have limited precision and cannot reliably be compared for equality.

Before to go deep into details, you need to be aware of some underlying concepts in Python:

- *First*, Python variables are just names. You create variable names by assigning them to objects that are instances of types. Variables do not, of themselves, have a type; it is the object to which they are bound that has a type. The name is just a label and, as such, it can be reassigned to a completely different object.
- *Second*, Python groups types according to how you can use them. For example, all types are either categorized as *mutable* or *immutable*. If a type is immutable, it means you can't change an object of that type once it's created. You can create a new data item and assign it to the same variable, but you cannot change the original immutable value.
- *Third*, Python also supports several collection types, sometimes referred to as sequences or data containers. Sequences share a common set of operations, although not all sequences support all of the operations.
- And *fourth*, some Python data types are *callable*. That means you can use the type name like a function to produce a new instance of the type. If no value is given, a default value is returned. You will see examples of this behavior in the following descriptions of the individual data types.

On the other hand, Python has a huge number of libraries included with the distribution. To keep things simple, most of these variables and functions are not accessible in a Python session. Instead, you have to **import** the name.

For example, there is a **math** module containing many useful functions. To access, say, the square root function, you can either first

```
from math import sqrt
```

and then, use the method or function as:

```
In [8]: from math import sqrt
        sqrt(10000)
```

```
Out[8]: 100.0
```

or you can simply **import** the full version of the **math** library itself

```
In [9]: import math
        print
        print dir(math)
```

```
['__doc__', '__file__', '__name__', '__package__', 'acos', 'acosh', 'asin', 'asinh', 'atan', 'a
```

```
In [10]: print
         print 'Sqrt of 1000 is ', math.sqrt(1000)
```

```
Sqrt of 1000 is  31.6227766017
```

You can define variables using the equals (=) sign:

```
In [11]: width = 20.
         length = 30

         print 'Type width: ',type(width)
         print 'Type lenght: ',type(length)
         area = length*width
         print
         print 'Area: ', area
         print 'Type Area: ', type(area)
```

```
Type width:  <type 'float'>
Type lenght:  <type 'int'>

Area:  600.0
Type Area:  <type 'float'>
```

Notice that output, even it has type float, it can be presented with different formats. Check below how to define the format of the outputs:

```
In [12]: from math import pi
         print "Pi as a decimal = %d" % pi
         print "Pi as a float = %f" % pi
         print "Pi with 4 decimal places = %.4f" % pi
         print "Pi with overall fixed length of 10 spaces, with 6 decimal places = %10.6f" % pi
         print "Pi as in exponential format = %e" % pi
```

```
Pi as a decimal = 3
Pi as a float = 3.141593
Pi with 4 decimal places = 3.1416
Pi with overall fixed length of 10 spaces, with 6 decimal places =   3.141593
Pi as in exponential format = 3.141593e+00
```

3.2 Some Pitfalls in Numerical Computing

In scientific computing, we never expect to get the exact answer. Inexactness is practically the definition of scientific computing. Getting the exact answer, generally with integers or rational numbers, is symbolic computing, an interesting but distinct subject. Suppose we are trying to compute the number x. The computer will produce an approximation, which we call $\hat{x}$. This $\hat{x}$ may agree with x to 16 decimal places, but the identity $x = \hat{x}$ (almost) never is true in the mathematical sense, if only because the computer does not have an exact representation for x.

Four primary sources of error are:

- **roundoff error** and floating-point arithmetics,
- **truncation error**,
- **termination of iterations**, and
- **statistical error** in Monte Carlo.

We will estimate the sizes of these errors, either a priori from what we know in advance about the solution, or a posteriori from the computed (approximate) solutions themselves. Software development requires distinguishing these errors from those caused by outright bugs. In fact, the bug may not be that a formula is wrong in a mathematical sense, but that an approximation is not accurate enough.

Here we will provide examples of the first two classes of errors. The third and forth will be explained in further chapters.

3.2.1 Roundoff error and floating-point arithmetics

Please, pay attention to the following examples. They do not need any further explanation.

```
In [13]: # Example 1.a

         a = 1
         b = 2
         print (a/b)*b
```

```
0
```

```
In [14]: # Example 1.b

         a = float(1)
         b = float(2)
         print (a/b)*b
```

```
1.0
```

```
In [15]: # Example 2.a

         x = 1/2
         y = 1/2.
         x==y

Out[15]: False

In [16]: # Example 2.b

         a = 1
         b = 2

         x = float(1)
         y = float(2)

         (a/b)==(x/y)

Out[16]: False

In [17]: # Example 3

         x = float(10)
         y = float(10000)
         print np.exp(x)*np.exp(-x)
         print np.exp(y)*np.exp(-y)
```

```
1.0
nan
```

Floating-point arithmetic on digital computers is inherently inexact. The 24-bits (including the hidden bit) of mantissa in a 32-bit floating-point number represent approximately 7 significant decimal digits. Unlike the real number system, which is continuous, a floating-point system has gaps between each number. If a number is not exactly representable, then it must be approximated by one of the nearest representable values.

```
In [18]: def add(a, b):
             if a + b == 0.3:
                 print('the sum was 0.3')
             else:
                 print ('{:0.18g} does not equal to 0.3'.format(a + b))


         add(0.1, 0.2)
```

```
0.300000000000000044 does not equal to 0.3
```

3.2.2 Truncation errors

Truncation errors are committed when an iterative method is terminated or a mathematical procedure is approximated, and the approximate solution differs from the exact solution. Similarly, discretization induces a discretization error because the solution of the discrete problem does not coincide with the solution of the continuous problem. For instance, in the iteration in the sidebar to compute the solution of $3x^3 + 4 = 28$, after 10 or so iterations, we conclude that the root is roughly 1.99 (for example). We therefore have a truncation error of 0.01.

Once an error is generated, it will generally propagate through the calculation. For instance, we have already noted that the operation + on a calculator (or a computer) is inexact. It follows that a calculation of the type `a+b+c+d+e` is even more inexact.

If you are in a situation where you care which way your decimal halfway-cases are rounded, you should consider using the decimal module. Incidentally, the decimal module also provides a nice way to "see" the exact value that's stored in any particular Python float

```
In [19]: from decimal import Decimal
         from math import sqrt
         x = sqrt(2)
         print
         print 'Root Square of 2 is: ', x
         print 'Root Square of 2 is: ', Decimal(x)
         print
         print 'Are both numbers the same number for the computer? ', x == Decimal(x)
         print 'But, Are the square of both the same for the computer? ', x**2 == (Decimal(x))*
```

```
Root Square of 2 is:  1.41421356237
Root Square of 2 is:  1.4142135623730951454746218587388284504413604736328125

Are both numbers the same number for the computer?  True
But, Are the square of both the same for the computer?  False
```

Another example of truncation error is that since 0.1 is not exactly 1/10, summing ten values of 0.1 may not yield exactly 1.0, either:

```
In [20]: summ = 0.0
         for __ in range(10):
             summ += 0.1

         print summ,'is for Python', Decimal(summ)
```

```
1.0 is for Python 0.99999999999999988897769753748434595763683319091796875
```

```
In [21]: vals = []
         for i in range(1, 10):
             number = 9.0 * 10.0 ** -i
             vals.append(number)
             total = sum(vals)
             expected = 1.0 - (1.0 * 10.0 ** i)
             diff = total - expected
             print '%2d  %22.21f  %22.21f' % (i, total, total-expected)
```

```
1  0.900000000000000022204  9.900000000000000355271
 2  0.989999999999999991118  99.989999999999994884092
 3  0.998999999999999999112  999.999000000000023646862
 4  0.999900000000000011013  9999.999900000000707223080
 5  0.999990000000000045510  99999.999989999996614642441
 6  0.999999000000000082267  999999.999998999992385506630
 7  0.999999900000000052636  9999999.999999899417161941528
 8  0.999999990000000060775  99999999.999999985098838806152
 9  0.999999999000000028282  1000000000.000000000000000000000
```

```
In [22]: import math
         def isSquare(x):
             return pow(int(math.sqrt(x)),2)-x == 0
```

CHAPTER

FOUR

COLLECTIONS AND DATA STRUCTURES

Python supports many powerful data structures. Superficially, these look like their counterparts in other programming languages, but in Python they often come with steroids. Everything in Python is an object and, therefore, has methods. This means that you can perform a host of operations on any variable. The built-in `dir()` and `help()` functions will reveal all. In this section you look at the standard data collections like:

- **Strings**
- **Lists**
- **Tuples**
- **Dictionaries**

and their most important operations.

4.1 Strings

Strings are lists of printable characters, and to create string literals, enclose them in single, double, or triple quotes as follows:

```
In [23]: print 'Hello, Dusseldorf!'
         print "Hello, Dusseldorf!!"
         print '''Hello, Dusseldorf!!!'''
```

```
Hello, Dusseldorf!
Hello, Dusseldorf!!
Hello, Dusseldorf!!!
```

The same type of quote used to start a string must be used to terminate it. Triple-quoted strings capture all the text that appears prior to the terminating triple quote, as opposed to single- and double-quoted strings, which must be specified on one logical line. Triple-quoted strings are useful when the contents of a string literal span multiple lines of text.

Strings are stored as sequences of characters indexed by integers, starting at zero.To extract a single character, use the indexing operator `s[i]` like this:

```
In [24]: greeting = "Hello, Dusseldorf!"
         print greeting
         print
         print greeting[0:5]
         print
         for i in xrange(len(greeting)):
             print (greeting[i])
```

```
Hello, Dusseldorf!

Hello

H
e
```

```
l
l
o
,

D
u
s
s
e
l
d
o
r
f
!
```

```
In [25]: # Properties of the string

         print 'Type of variable:',type(greeting)
         print 'Lenght of variable:',len(greeting)
```

```
Type of variable: <type 'str'>
Lenght of variable: 18
```

```
In [26]: # Find and replace in the string:

         greeting = greeting.replace('Dusseldorf','Cologne')
         print 'Replacing Dusseldorf: ', greeting
         print 'And Cologne is in position', greeting.find('Cologne'), 'in the string'
```

```
Replacing Dusseldorf:  Hello, Cologne!
And Cologne is in position 7 in the string
```

```
In [27]: # An interesting example... is like a list of a list (see next section about lists)

         statement = ['Hello', 'Dusseldorf']
         print
         print 'Yes, I am in',statement[1],'...'
         print
         print 'and now, I am in',greeting[7:-1]
```

```
Yes, I am in Dusseldorf ...

and now, I am in Cologne
```

And, remember always that, in case that you need help…

```
In [28]: help(len)

         # or in iPython

         len?
```

```
Help on built-in function len in module __builtin__:

len(...)
    len(object) -> integer

    Return the number of items of a sequence or collection.
```

4.2 Lists

Lists are sequences of arbitrary objects, or in other words, a container of objects. This is just a way to collect all type of objects in a container that is *mutable*.

You create a list by enclosing values in square brackets, as follows:

```
In [29]: days_of_the_week = ["Sunday","Monday","Tuesday","Wednesday","Thursday","Friday","Satur
         print
         print days_of_the_week
```

```
['Sunday', 'Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday']
```

Lists are indexed by *integers*, starting with *zero*. Use the indexing operator to access and modify individual items of the list:

```
In [30]: print
         print 'First days of the week: ', days_of_the_week[0]
         print
         print 'Last day of the week: ', days_of_the_week[-1]
         print
         print 'Weekdays: ', days_of_the_week[1:6]
```

```
First days of the week:  Sunday

Last day of the week:  Saturday

Weekdays:  ['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday']
```

Additionally, we can iteratively retrieve the different elements of the list:

```
In [31]: for i, day in enumerate(days_of_the_week):
             print i, '-> ', day
```

```
0 ->  Sunday
1 ->  Monday
2 ->  Tuesday
3 ->  Wednesday
4 ->  Thursday
5 ->  Friday
6 ->  Saturday
```

An **empty** list can be created, and be filled or deleted sequentially later:

```
In [32]: empty_list = []
         print empty_list
```

```
[]
```

```
In [33]: empty_list.append('no so empty now')
         print empty_list
```

```
['no so empty now']
```

```
In [34]: del empty_list[0]
         print empty_list
```

```
[]
```

As we mentioned in the definition of list, it can contain any kind of Python object, including other lists, as in the following example:

```
In [35]: a = [1,"Dave",3.14, ["Mark", 7, 9, [100,101]], 10]
         print
         print a
```

```
[1, 'Dave', 3.14, ['Mark', 7, 9, [100, 101]], 10]
```

The `range(start, stop, step)` and `xrange(start, stop, step)` commands are a convenient and quick way to make **sequential lists** of integers numbers:

```
In [36]: evens = range(0,20,2)
         evens_reversed = range(20,0,-2)

         print
         print evens
         print
         print evens_reversed
```

```
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]

[20, 18, 16, 14, 12, 10, 8, 6, 4, 2]
```

Basic operations with lists: length, concatenation, repetition, membership and iteration

```
In [37]: # 1. Lenght

         languages = ["Fortran","C","C++"]

         print
         print 'languages =', languages
         print
         print 'lenght of the list:', len(languages)
```

```
languages = ['Fortran', 'C', 'C++']

lenght of the list: 3
```

```
In [38]: # We can append new elements in the list, and check the lenght

         languages.append("Python")
         languages.append(["Haskell","Lisp"])

         print
         print 'languages =', languages
         print
         print 'lenght of the list:', len(languages)
```

```
languages = ['Fortran', 'C', 'C++', 'Python', ['Haskell', 'Lisp']]

lenght of the list: 5
```

```
In [39]: # Removing the last element in the list of languages

         languages.remove(languages[len(languages)-1])
         print
         print 'Last two languages:',languages[len(languages)-2:]
```

```
Last two languages: ['C++', 'Python']
```

```
In [40]: # 2. Concatenation

         languages1 = ["Fortran","C","C++", "Python"]
         languages2 = ["Haskell","Lisp"]

         languages = languages1 + languages2 + ['Scala']

         print
         print languages
```

```
['Fortran', 'C', 'C++', 'Python', 'Haskell', 'Lisp', 'Scala']

In [41]: # 3. Repetition

         print languages * 4

['Fortran', 'C', 'C++', 'Python', 'Haskell', 'Lisp', 'Scala', 'Fortran', 'C', 'C++', 'Python',

In [42]: # 4. Memebership

         'Python' in languages

Out[42]: True

In [43]: 'Java' in languages

Out[43]: False

In [44]: 'Python' or 'Java' in languages

Out[44]: 'Python'

In [45]: 'Python' and 'Java' in languages

Out[45]: False

In [46]: # 5. Iteration

         for i, language in enumerate(languages): print '{i}.- Language name: {lang}'.format(i

1.- Language name: Fortran
2.- Language name: C
3.- Language name: C++
4.- Language name: Python
5.- Language name: Haskell
6.- Language name: Lisp
7.- Language name: Scala
```

List Comprenhesion, Iterators, Generators and Operating among Lists (Advance)

```
In [47]: #TODO
```

Sometimes, instead to use an NumPy array or a vector, we just want to operate `(+, - ...)` among the elements of the list. In order to du that we have to create our own list class and overload the operator using __infix sintax:

```
In [48]: class mylist(list):
             def __init__(self, *args):
                 super(mylist, self).__init__(args)

             def __sub__(self, other):
                 return self.__class__(*[item for item in self if item not in other])

In [49]: x = mylist(1,2,3,4)
         y = mylist(2,5,2)

         z = x-y

         print z

         print type (z)

[1, 3, 4]
<class '__main__.mylist'>
```

4.3 Tuples

A **tuple** is a sequence object like a list or a string. It's constructed by grouping a sequence of objects together with commas, either without brackets, or with parentheses. Tuples are *inmutable*, meaning that they cannot be modified once created (you can't append to them or change the elements of them):

```
In [50]: languages_t = ("Fortran","C","C++", "Python")

         print
         print languages_t
         print
         print 'Size:',len(languages_t),'||','Type:',type(languages_t)
```

```
('Fortran', 'C', 'C++', 'Python')

Size: 4 || Type: <type 'tuple'>
```

```
In [51]: # Unpacking
         w,x,y,z = languages_t
         print z
```

```
Python
```

Tuples respond to the + and ⋆ operators much like strings or lists; they mean concatenation and repetition here too, except that the result is a new tuple, not a list or string, preserving the type of the object.

```
In [52]: # 1. Length
         print 'Size:',len(languages_t)
```

```
Size: 4
```

```
In [53]: #2. Concatenation
         languages_t + ('Java','Haskell')

Out[53]: ('Fortran', 'C', 'C++', 'Python', 'Java', 'Haskell')

In [54]: #3. Repetition
         print ('Java','Haskell')*3
         print languages_t *2
```

```
('Java', 'Haskell', 'Java', 'Haskell', 'Java', 'Haskell')
('Fortran', 'C', 'C++', 'Python', 'Fortran', 'C', 'C++', 'Python')
```

```
In [55]: #4. Membership
         'Python' in languages_t

Out[55]: True

In [56]: #5. Iteration
         print z[:]
         print
         for i,letter in enumerate(z):
             print z[i], '=', letter
```

```
Python

P = P
y = y
t = t
h = h
o = o
n = n
```

Tuples are useful anytime you want to group different pieces of data together in an object, but don't want to create a full-fledged class (see below) for them. For example, let's say you want the Cartesian coordinates of some objects in your program.

4.4 Dictionaries

A **dictionary** is another container type that can store any number of Python objects, including other container types. Dictionaries consist of pairs (called items) of keys and their corresponding values. A dictionary is *mutable* and sometimes it is also called *associative arrays* or *hash maps* in other languages.

Each key is separated from its value by a colon `{"key":value}`, the items are separated by commas, and the whole thing is enclosed in curly braces. An empty dictionary without any items is written with just two curly braces, like this: `{}`.

Keys are unique within a dictionary while values may not be. The values of a dictionary can be of any type, but the keys must be of an immutable data type such as strings, numbers, or tuples.

```
In [57]: ages = {"Rick": 46, "Bob": 86, "Fred": 21}
         print
         print "Rick's age is ",ages["Rick"]
```

```
Rick's age is  46
```

```
In [58]: # Alternative way to store a dictionary
         ages2 = dict(Rick=46, Bob=86, Fred=20)
         print
         print "Rick's age is ",ages2["Rick"]
```

```
Rick's age is  46
```

```
In [59]: data = {}
         data['k1'] = True
         data['k2'] = 2
         data['k3'] = 3.0
         print
         print data
         print
         print 'Size:',len(data),'||','Type:',type(data)
```

```
{'k3': 3.0, 'k2': 2, 'k1': True}

Size: 3 || Type: <type 'dict'>
```

```
In [60]: # Add-in a new element in the list
         data['k4'] = [100, 101, 102]
         print
         print 'New data:',data['k4'], 'in position', data.keys()[-1]
         print
         print data
```

```
New data: [100, 101, 102] in position k4

{'k3': 3.0, 'k2': 2, 'k1': True, 'k4': [100, 101, 102]}
```

```
In [61]: # Getting data using the label or index
         x = 'k4'
         print
         print 'Retriving from ',x, 'the data', data.get('k4')
```

```
Retriving from  k4 the data [100, 101, 102]
```

4.5 Hash Functions (Advance)

Given a collection of items, a **hash function** that maps each item into a unique slot is referred to as a **perfect hash function**. If we know the items and the collection will never change, then it is possible to construct a perfect hash function (refer to the exercises for more about perfect hash functions). Unfortunately, given an arbitrary collection of items, there is no systematic way to construct a perfect hash function.

One way to always have a perfect hash function is to increase the size of the hash table so that each possible value in the item range can be accommodated. This guarantees that each item will have a unique slot. Although this is practical for small numbers of items, it is not feasible when the number of possible items is large. For example, if the items were nine-digit Social Security numbers, this method would require almost one billion slots. If we only want to store data for a class of 25 students, we will be wasting an enormous amount of memory.

Our goal is to create a hash function that minimizes the number of collisions, is easy to compute, and evenly distributes the items in the hash table. There are a number of common ways to extend the simple remainder method.

The **folding method** for constructing hash functions begins by dividing the item into equal-size pieces (the last piece may not be of equal size). These pieces are then added together to give the resulting hash value. For example, if our item was the phone number `436-555-4601`, we would take the digits and divide them into groups of 2 (`43,65,55,46,01`). After the addition, `43+65+55+46+01`, we get `210`. If we assume our hash table has `11` slots, then we need to perform the extra step of dividing by `11` and keeping the remainder. In this case `210 % 11` is 1, so the phone number `436-555-4601` hashes to slot 1. Some folding methods go one step further and reverse every other piece before the addition. For the above example, we get `43+56+55+64+01=219` which gives `219 % 11=10`.

We can also create hash functions for character-based items such as strings. The word `"abcdef"` can be thought of as a sequence of ordinal values.We can then take these six ordinal values, add them up, and use the remainder method to get a hash value:

```
In [62]: def hash(astring, tablesize):
             sum = 0
             for pos in range(len(astring)):
                 sum = sum + ord(astring[pos])

             return sum%tablesize
```

```
In [63]: print hash('abcdef',11)
         print hash('lmnopq',11)
```

```
3
3
```

```
In [64]: mydict = {str(hash('abcdef',11)):'abcdef',
                   str(hash('ghijk',11)):'ghijk',
                   str(hash('lmnopq',11)):'lmnopq'}
         '''
         Notice the collision in the slot 3:
         because str(hash('lmnopq',11))==str(hash('abcdef',11))
         '''
         print
         print mydict
```

```
{'8': 'ghijk', '3': 'lmnopq'}
```

Another numerical technique for constructing a hash function is called the **mid-square method**. We first square the item, and then extract some portion of the resulting digits. For example, if the item were `44`, we would first compute `44^2=1,936`. By extracting the middle two digits, `93`, and performing the remainder step, we get `5 (93 % 11)`.

The important thing to remember is that the hash function has to be efficient so that it does not become the dominant part of the storage and search process. If the hash function is too complex, then it becomes more work to compute the slot name than it would be to simply do a basic sequential or binary search as described earlier. This would quickly defeat the purpose of hashing.

See here below a full implementation of a **Hash Table** class with a **hash function** as a method:

```
In [65]: class HashTable(object):
             def __init__(self, size):
                 self.size = size
```

```
        self.slots = [None] * self.size
        self.data = [None] * self.size

    def __getitem__(self, key):
        return self.get(key)

    def __setitem__(self, key, data):
        self.put(key, data)

    def hashfunction(self, key, size):
        return key%size

    def rehash(self, oldhash, size):
        return (oldhash + 1)%size

    def put(self, key, data):

        hashvalue = self.hashfunction(key, len(self.slots))

        if self.slots[hashvalue] == None:
            self.slots[hashvalue] = key
            self.data[hashvalue] = data
        else:
            if self.slots[hashvalue] == key:
                self.data[hashvalue] = data  # replacing data (collision case)
            else:
                nextslot = self.rehash(hashvalue, len(self.slots))
                while self.slots[nextslot] != None and self.slots[nextslot] != key:
                    nextslot = self.rehash(nextslot, len(self.slots))

                if self.slots[nextslot] == None:
                    self.slots[nextslot] = key
                    self.data[nextslot] = data
                else:
                    self.data[nextslot] = data # replace

    def get(self,key):
        startslot = self.hashfunction(key,len(self.slots))
        data = None
        stop = False
        found = False
        position = startslot
        while self.slots[position] != None and not found and not stop:
            if self.slots[position] == key:
                found = True
                data = self.data[position]
            else:
                position=self.rehash(position,len(self.slots))
                if position == startslot:
                    stop = True

        return data
```

```
In [66]: H=HashTable(10)

         H[54]="cat"
         H[26]="dog"
         H[93]="lion"
         H[17]="tiger"
         H[77]="bird"
         H[31]="cow"
         H[44]="goat"
         H[55]="pig"
         H[20]="chicken"

         print 'Slots: ',H.slots
         print 'Data: ', H.data
```

```
Slots:  [20, 31, None, 93, 54, 44, 26, 17, 77, 55]
```

```
Data:  ['chicken', 'cow', None, 'lion', 'cat', 'goat', 'dog', 'tiger', 'bird', 'pig']
```

CHAPTER

FIVE

CONTROL FLOW

This chapter describe some of the most basic Python statements that help to control the flow of the computational work inside any algorithm.

- `if`, `elif` and `else`
- `for` loops
- `while` loops
- `exception` handling
- Iteration, Indentation and Blocks
- Comprehension

5.1 `if`, `elif`, and `else`

```
In [67]: car = 10
         bus = 2.5
         cash = 1.5

         if car < cash:
             print 'I can drive'

         if bus < cash and car > cash:
             print "I'll take the bus"

         if bus > cash:
             print 'Walking.'

Walking.
```

```
In [68]: if car < cash:
             print 'Enjoy your car ride'
         elif bus < cash:
             print 'Another one rides the bus'
         else:
             print 'Walking..'

Walking..
```

```
In [69]: action = 'car' if car < cash else 'Walking...' #one-liner

         print action

Walking...
```

5.2 The `for` loop

```
In [70]: numbers = [1,2,3,4,5]

         for i in numbers:
             print i,

         print '  '

         for num in numbers:
             print num,

1 2 3 4 5
1 2 3 4 5
```

```
In [71]: for i in xrange(10):
             print i,

0 1 2 3 4 5 6 7 8 9
```

```
In [72]: for letter in "Sunday":
             print letter

S
u
n
d
a
y
```

```
In [73]: for i in xrange(10):
             print i, i%2
             if i % 2:
                 continue
             print "still here",

0 0
still here 1 1
2 0
still here 3 1
4 0
still here 5 1
6 0
still here 7 1
8 0
still here 9 1
```

```
In [74]: for i in xrange(10):
             if i == 5:
                 break
             print i,

0 1 2 3 4
```

```
In [75]: index = 0
         for day in days_of_the_week:
             print index, days_of_the_week[index], day
             index += 1

0 Sunday Sunday
1 Monday Monday
2 Tuesday Tuesday
3 Wednesday Wednesday
4 Thursday Thursday
5 Friday Friday
6 Saturday Saturday
```

```
In [76]: for i, day in enumerate(days_of_the_week):
             print i, days_of_the_week[i], day

0 Sunday Sunday
1 Monday Monday
2 Tuesday Tuesday
3 Wednesday Wednesday
4 Thursday Thursday
5 Friday Friday
6 Saturday Saturday
```

5.3 `while` loop

```
In [77]: count = 0
         while (count < 10):
             print count,
             count += 1

0 1 2 3 4 5 6 7 8 9
```

5.4 `exception` handling

Python is an interpreted language, which means that there is no compiler to compile your code and find any logic or syntax errors before you run it. So how does Python handle this? Python uses **exceptions** to handle errors.

For example, if you try the following code in your interpreter, you will get an `ValueError`:

```
In [128]: num = '123.4d'
          print float(num)

        ---------------------------------------------------------------------------

        ValueError                                Traceback (most recent call last)

        <ipython-input-128-320d5b550040> in <module>()
          1 num = '123.4d'
    ----> 2 print float(num)


        ValueError: invalid literal for float(): 123.4d
```

As you can see, when you try to pass a string to a mathematic function, which can only operate on integers and/or floats, it throws a `ValueError`. This tells you that the data you sent to the function is not of the correct type. Because Python is not a strongly typed language, nor is it compiled, the only errors that you will get are exceptions, which will crop up at run time. When an `exception` is thrown at run time, your entire program will quit if there is no exception handling in place. An `exception` is an object like any other Python object, and when converted to a string (e.g., when printed), the `exception` produces a message text.

The most important way to manage this error are the `try?except` block will try a piece of code and if the code throws one of the preceding exceptions, it will catch that `exception` and print out an `error message`, as defined in the base exception class, or you can even print your own error messages for each `exception`:

```
In [129]: try:
              print float(num)
          except (ValueError, TypeError),e:
              print e
              print 'DO SOMETHING ELSE HERE'
```

```
invalid literal for float(): 123.4d
DO SOMETHING ELSE HERE
```

The logic works like this. If the statements in the `try` block's suite all execute without raising an exception, the `except` blocks are skipped. If an `exception` is raised inside the `try` block, control is immediately passed to the suite corresponding to the first matching exception. This means that any statements in the suite that follow the one that caused the exception will not be executed.

```
In [130]: values = [1,2,3]
          try:
              print float(values)
          except ValueError:
              print "nope"
          except TypeError:
              print float(values[0])
```

```
1.0
```

```
In [131]: try:
              print float("454c")
              print float([1,2,3])
          except:
              print "error"
```

```
error
```

5.5 Iteration, Indentation, and Blocks

One of the most useful things you can do with lists is to *iterate* through them, i.e. to go through each element one at a time. To do this in Python, we use the **for** statement:

```
In [132]: for day in days_of_the_week:
              print day
```

```
Sunday
Monday
Tuesday
Wednesday
Thursday
Friday
Saturday
```

This code snippet goes through each element of the list called **days_of_the_week** and assigns it to the variable **day**. It then executes everything in the indented block (in this case only one line of code, the print statement) using those variable assignments. When the program has gone through every element of the list, it exists the block.

Every programming language defines blocks or scopes of code in some way. In Fortran, one uses END statements (ENDDO, ENDIF, etc.) to define code blocks. In C, C++, and Perl, one uses curly braces {} to define the scope or blocks.

Python uses a colon (":"), followed by **indentation level** to define code blocks. Everything at a higher level of indentation is taken to be in the same block.

```
In [133]: for day in days_of_the_week:
              statement = "Today is " + day
              print statement
```

```
Today is Sunday
Today is Monday
Today is Tuesday
Today is Wednesday
Today is Thursday
Today is Friday
Today is Saturday
```

```
In [134]: for i in range(20):
              print "The square of ",i," is ",i*i
```

```
The square of  0  is  0
The square of  1  is  1
The square of  2  is  4
The square of  3  is  9
The square of  4  is  16
The square of  5  is  25
The square of  6  is  36
The square of  7  is  49
The square of  8  is  64
The square of  9  is  81
The square of  10  is  100
The square of  11  is  121
The square of  12  is  144
The square of  13  is  169
The square of  14  is  196
The square of  15  is  225
The square of  16  is  256
The square of  17  is  289
The square of  18  is  324
The square of  19  is  361
```

5.6 Introduction to List Comprehension

As you probably have noticed in the previous section, writing a piece of code such as this, is slow and can be more efficient:

```
In [135]: numbers = range(10)
          size = len(numbers)
          evens = []
          i = 0
          while i < size:
              if i % 2 == 0:
                  evens.append(i)
              i += 1

          print evens
```

```
[0, 2, 4, 6, 8]
```

This may work for C, but it actually makes things slower for Python because:

- It makes the interpreter work on each loop to determine what part of the sequence has to be changed.
- It makes you keep a counter to track what element has to be treated.

A list comprehension is the correct answer to this pattern. It uses wired features that automate parts of the previous syntax:

```
In [136]: [i for i in range(10) if i % 2 == 0]
```

```
Out[136]: [0, 2, 4, 6, 8]
```

Besides the fact that this writing is more efficient, it is way shorter and involves fewer elements. In a bigger program, this means less bugs and code that is easy to read and understand.

As mentioned early in this book, Python supports the procedural, object-oriented, and function programming paradigms. In fact, Python has a host of tools that most would considered functional in nature like closures, generators, lambdas, comprehensions, maps, decorators, function objects, and more.

Let us work through another example to see the basics. Just notice that Python's built-in function `ord` returns the integer code point of a single character

```
In [137]: ord('s')

Out[137]: 115

In [138]: res = []
          for x in 'spam':
              res.append(ord(x))

          print 'Word spam in integers: ', res

Word spam in integers:  [115, 112, 97, 109]


In [139]: # more functional oriented implementation:

          res = list(map(ord, 'spam'))
          print 'Word spam in integers: ', res

Word spam in integers:  [115, 112, 97, 109]


In [140]: # a list comprenhension implementation:

          res = [ord(x) for x in 'spam']
          print 'Word spam in integers: ', res

Word spam in integers:  [115, 112, 97, 109]
```

List comprehensions become more convenient, though, when we wish to apply an arbitrary expression to an iterable instead of a function:

```
In [141]: [x ** 2 for x in range(10)]

Out[141]: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

In [142]: list(map((lambda x: x ** 2), range(10)))

Out[142]: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

List comprehensions are even more general than shown so far. List comprehensions with if clauses can be thought of as analogous to the filter built-in

```
In [143]: [x**2 for x in range(10) if x % 2 == 0]

Out[143]: [0, 4, 16, 36, 64]

In [144]: list( map((lambda x: x**2), filter((lambda x: x % 2 == 0), range(10))) )

Out[144]: [0, 4, 16, 36, 64]
```

One basic way to code matrixes (a.k.a. multidimensional arrays) in Python is with nested list structures. The following, for example, defines two 3 × 3 matrixes as lists of nested lists:

```
In [145]: M = [[1, 2, 3],
               [4, 5, 6],
               [7, 8, 9]]

          N = [[2, 2, 2],
               [3, 3, 3],
               [4, 4, 4]]

          print M[1]
          print M[1][2]
```

```
[4, 5, 6]
6
```

```
In [146]: [row[1] for row in M] # Column 2

Out[146]: [2, 5, 8]

In [147]: [M[row][1] for row in (0, 1, 2)] # Using offsets

Out[147]: [2, 5, 8]

In [148]: [M[i][i] for i in range(len(M))] # Diagonals

Out[148]: [1, 5, 9]

In [149]: [M[i][len(M)-1-i] for i in range(len(M))]

Out[149]: [3, 5, 7]
```

5.7 Code Example: The Fibonacci Sequence

The Fibonacci sequence is a sequence of numbers that starts with 0 and 1, and then each successive entry is the sum of the previous two. Thus, the sequence goes `0,1,1,2,3,5,8,13,21,34,55,89,...`

A very common exercise in programming books is to compute the Fibonacci sequence up to some number **n**. Please, notice the comments below.

```
In [150]: n = 20
          sequence = [0,1]
          for i in range(2,n): # This is going to be a problem if we ever set n <= 2!
              sequence.append(sequence[i-1]+sequence[i-2])
          print sequence

[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181]
```

Comments to the code above

- *First*, we define the variable **n**, and set it to the integer 20. **n** is the length of the sequence we're going to form, and should probably have a better variable name.
- *Second*, we create a variable called **sequence**, and initialize it to the list with the integers 0 and 1 in it, the first two elements of the Fibonacci sequence. We have to create these elements "by hand", since the iterative part of the sequence requires two previous elements.
- *Third*, we then have a for loop over the list of integers from 2 (the next element of the list) to **n** (the length of the sequence). Notice that after the colon, we see a hash tag "#", and then a **comment** that if we had set **n** to some number less than 2 we would have a problem. Comments in Python start with #, and are good ways to make notes to yourself or to a user of your code explaining why you did what you did. Better than the comment here would be to test to make sure the value of **n** is valid, and to complain if it isn't; we'll try this later.
- *Fourth*, in the body of the loop, we append to the list an integer equal to the sum of the two previous elements of the list.
- *And finally*, exiting the loop (ending the indentation) we then print out the whole list.

CHAPTER

SIX

FUNCTIONS

6.1 The definition of Function

We might want to use the Fibonacci snippet, showed at the end of the previous chapter, with different sequence lengths. We could cut an paste the code into another cell, changing the value of **n**, but it's easier and more useful to make a function out of the code. We do this with the **def** statement in Python:

```
In [151]: def fibonacci(sequence_length):
              "Return the Fibonacci sequence of length *sequence_length*"
              sequence = [0,1]
              if sequence_length < 1:
                  print "Fibonacci sequence only defined for length 1 or greater"
                  return
              if 0 < sequence_length < 3:
                  return sequence[:sequence_length]
              for i in range(2,sequence_length):
                  sequence.append(sequence[i-1]+sequence[i-2])
              return sequence
```

A **function** in Python is defined using the keyword `def`, followed by a function name, a signature within parentheses `()`, and a colon `:`. The previous code, with one additional level of indentation, is the function body, and can be called as:

```
In [152]: fibonacci(12)

Out[152]: [0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]

In [153]: help(fibonacci)

Help on function fibonacci in module __main__:

fibonacci(sequence_length)
    Return the Fibonacci sequence of length *sequence_length*
```

Functions can also call themselves, something that is often called **recursion**. Here you have an example of recursion by computing the factorial function. The factorial is defined for a positive integer **n** as

$$n! = n(n-1)(n-2)\cdots 1$$

First, note that we do not need to write a function at all, since this is a function built into the standard math library. However, if we did want to write a function ourselves, we could do recursively by:

```
In [154]: def fact(n):
              if n <= 0:
                  return 1
              return n*fact(n-1)

In [155]: fact(100)

Out[155]: 9332621544394415268169923885626670049071596826438162146859296389521759999322991560894
```

We can return multiple values from a function using tuples (see above):

```
In [156]: def powers(x):
              """
              Return a few powers of x.
              """
              return x ** 2, x ** 3, x ** 4
```

```
In [157]: powers(90)
```

```
Out[157]: (8100, 729000, 65610000)
```

```
In [158]: x2, x3, x4 = powers(3)

          print(x3)
```

```
27
```

```
In [159]: x2, x3, x4 = powers(3)

          print(x3)
```

```
27
```

As we have seen, **functions** are a means by which we can package up and parameterize functionality. Four kinds of functions can be created in Python: **global functions**, **local functions**, **lambda functions**, and **methods**.

Every function we have created so far has been a **global function**. Global objects (including functions) are accessible to any code in the same module (i.e., the same .py file) in which the object is created. Global objects can also be accessed from other modules.

Local functions (also sometimes called *nested functions*) are functions that are defined inside other functions. These functions are visible only to the function where they are defined; they are especially useful for creating small helper functions that have no use elsewhere.

Lambda functions are expressions, so they can be created at their point of use; however, they are much more limited than normal functions. In Python we can also create lambda functions, using the `lambda` keyword:

```
In [160]: f1 = lambda x: x**2

          # is equivalent to

          def f2(x):
              return x**2
```

```
In [161]: f1(2), f2(2)
```

```
Out[161]: (4, 4)
```

```
In [162]: # map is a built-in python function that allows us to treat function as objects...

          map(lambda x: x**2, range(-3,4))
```

```
Out[162]: [9, 4, 1, 0, 1, 4, 9]
```

Finally, **methods** are functions that are associated with a particular data type and can be used only in conjunction with the data type.

6.2 The Fibonacci sequence in a Generator (Advance)

Since Python 2.2, **generators** provide an elegant way to write simple and efficient code for functions that return a list of elements. Based on the `yield` directive, they allow you to pause a function and return an intermediate result. The function saves its execution context and can be resumed later if necessary.

For example, the *Fibonacci series* can be written with an iterator:

```
In [163]: def fibonacci_f():
              a, b = 0, 1
              while True:
                  yield b
                  a, b = b, a + b
```

This function returns a **generator object**, a special iterator, which knows how to save the execution context. It can be called indefinitely, yielding the next element of the suite each time. The syntax is concise, and the infinite nature of the algorithm does not disturb the readability of the code anymore. It does not have to provide a way to make the function stoppable. In fact, it looks similar to how the series would be designed in pseudo-code.

```
In [164]: fib = fibonacci_f()

In [165]: fib.next()

Out[165]: 1

In [166]: fib.next()

Out[166]: 1

In [167]: fib.next()

Out[167]: 2

In [168]: [fib.next() for i in range(10)]

Out[168]: [3, 5, 8, 13, 21, 34, 55, 89, 144, 233]
```

Generators should be considered every time you deal with a function that returns a sequence or works in a loop. Returning the elements one at a time can **improve the overall performance**, when they are passed to another function for further work.

In that case, the resources used to work out one element are most of the time less important than the resources used for the whole process. Therefore, they can be kept low, making the program more efficient. For instance, the Fibonacci sequence is infinite, and yet the generator that generates it does not require an infinite amount of memory to provide the values one at a time. A common use case is to stream data buffers with generators. They can be paused, resumed, and stopped by third-party code that plays over the data, and all the data need not be loaded before starting the process.

Generators can also help in breaking the complexity, and raising the efficiency of some data transformation algorithms that are based on several suites. Thinking of each suite as an iterator, and then combining them into a high-level function is a great way to avoid a big and unreadable function. Moreover, this can provide a live feedback to the whole processing chain.

In the example below, each function defines a transformation over a sequence. They are then chained and applied. Each call processes one element and returns its result:

```
In [169]: def power(values):
              for value in values:
                  print 'powering %s' % value
                  yield value

In [170]: def adder(values):
              for value in values:
                  print 'adding to %s' % value
                  if value % 2 == 0:
                      yield value + 3
                  else:
                      yield value + 2

In [171]: elements = [1, 4, 7, 9, 12, 19]

In [172]: result = adder(power(elements))

In [173]: result.next()

powering 1
adding to 1


Out[173]: 3
```

```
In [174]: result.next()

powering 4
adding to 4


Out[174]: 7
```

6.3 Explaining Paradigms: Procedural vs. OOP vs. Functional (Advance)

You have already notice that for some kinds of computations, we can ignore Python's Object-Oriented features and write simple numeric algorithms. For example, we might write something in **procedural style**, like the following to get the range of numbers:

```
In [175]: s = 0
          for n in range (1,10):
              if n%3 == 0 or n%5 == 0:
                  s+=n
          print s

23
```

We have made this program strictly **procedural**, avoiding any explicit use of Python's object features. The program's state is defined by the values of the variables `s` and `n`. The variable, `n`, takes on values such that `1 ≤ n < 10`. As the loop involves an ordered exploration of values of `n`, we can prove that it will terminate when `n == 10`. Similar code would work in C or Java using their primitive (non-object) data types.

We can exploit Python's **Object-Oriented Programming (OOP)** features and create a similar program:

```
In [176]: m = list()
          for n in range(1,10):
              if n % 3 == 0 or n % 5 == 0:
                  m.append(n)
          print sum(m)

23
```

This program produces the same result but it accumulates a stateful collection object, `m`, as it proceeds. The state of the computation is defined by the values of the variables `m` and `n`.

The syntax of `m.append(n)` and `sum(m)` can be confusing. It causes some programmers to insist (wrongly) that Python is somehow not purely Object-Oriented because it has a mixture of the `function()` and `object.method()` syntax. Rest assured, Python is purely Object-Oriented. Some languages, like C++, allow the use of primitive data type such as `int`, `float`, and `long`, which are not objects. Python doesn't have these primitive types. The presence of prefix syntax doesn't change the nature of the language.

To be *pedantic*, we could fully embrace the **Object-Oriented model**, the subclass, the `list` class, and add a `sum` method:

```
In [177]: class SummableList(list):
              def sum( self ):
                  s= 0
                  for v in self.__iter__():
                      s += v
                  return s
```

If we initialize the variable, `m`, with the `SummableList()` class instead of the `list()` method, we can use the `m.sum()` method instead of the `sum(m)` method. This kind of change can help to clarify the idea that Python is truly and completely object-oriented. The use of prefix function notation is purely syntactic sugar.

All three of these examples rely on variables to explicitly show the state of the program. They rely on the assignment statements to change the values of the variables and advance the computation toward completion. We can insert the assert statements throughout these examples to demonstrate that the expected state changes are implemented properly.

In a functional sense, the sum of the multiples of `3` and `5` can be defined in two parts:

- The sum of a sequence of numbers
- A sequence of values that pass a simple test condition, for example, being multiples of three and five

The sum of a sequence has a simple, recursive definition:

```
In [178]: def sum(seq):
              if len(seq) == 0: return 0
              return seq[0] + sum(seq[1:])
```

We have defined the sum of a sequence in two cases: the base case states that the sum of a zero length sequence is `0`, while the recursive case states that the sum of a sequence is the first value plus the sum of the rest of the sequence. Since the recursive definition depends on a shorter sequence, we can be sure that it will (eventually) devolve to the base case.

The + operator on the last line of the preceding example and the initial value of 0 in the base case characterize the equation as a sum. If we change the operator to `*` and the initial value to `1`, it would just as easily compute a product.

Similarly, a sequence of values can have a simple, recursive definition, as follows:

```
In [179]: def until(n, filter_func, v):
              if v == n: return []
              if filter_func(v): return [v] + until( n, filter_func, v+1 )
              else: return until(n, filter_func, v+1)
```

In this function, we have compared a given value, `v`, against the upper bound, `n`. If `v` reaches the upper bound, the resulting list must be empty. This is the base case for the given recursion.

There are two more cases defined by the given `filter_func()` function. If the value of `v` is passed by the `filter_func()` function, it will create a very small list, containing one element, and append the remaining values of the `until()` function to this list. If the value of `v` is rejected by the `filter_func()` function, this value is ignored and the result is simply defined by the remaining values of the `until()` function.

We can see that the value of `v` will increase from an initial value until it reaches `n`, assuring us that we will reach the base case soon.

Here it is how we can use the `until()` function to generate the multiples of 3 or 5. First, we define a handy `lambda` object to filter values:

```
In [180]: mult_3_5= lambda x: x%3==0 or x%5==0

          print
          print mult_3_5(3)
          print mult_3_5(5)

True
True
```

This function can be used with the `until()` function to generate a sequence of values, which are multiples of 3 or 5.

The `until()` function for generating a sequence of values works as follows:

```
In [181]: until(10, lambda x: x%3==0 or x%5==0, 0)

Out[181]: [0, 3, 5, 6, 9]
```

We can use our recursive `sum()` function to compute the sum of this sequence of values. The various functions, such as `sum()`, `until()`, and `mult_3_5()` are defined as simple recursive functions. The values are computed without restoring to use intermediate variables to store state.

A hybrid functional version might look like the following:

```
In [182]: sum([ n for n in range(1,10) if n%3==0 or n%5==0 ])
```

```
Out[182]: 23
```

We have used nested generator expressions to iterate through a collection of values and compute the sum of these values. The `range(1, 10)` method is an iterable and, consequently, a kind of generator expression: it generates a sequence of values $\{n|1 \leq n \leq 10\}$

The more complex expression, `n for n in range(1, 10) if n%3==0 or n%5==0`, is also an iterable expression. It produces a set of values $\{n|1 \leq n \leq 10 \wedge (n \mod 3 = 0 \vee n \mod 5 = 0)\}$

The `if` clause of the expression can be extracted into a separate function, allowing us to easily repurpose this with other rules. We could also use a higher-order function named `filter()` instead of the if clause of the generator expression.

6.4 A classic example of functional programming: the Newton-Raphson algorithm (Advance)

As part of our introduction, we examine a classic example of functional programming. This is based on the classic paper **Why Functional Programming Matters** by *John Hughes*. The article appeared in a paper called *Research Topics in Functional Programming*, edited by *D. Turner*, published by *Addison-Wesley* in 1990.

Here it is the link to the paper *Research Topics in Functional Programming*:

http://www.cs.kent.ac.uk/people/staff/dat/miranda/whyfp90.pdf

This discussion of functional programming in general is profound. There are several examples given in the paper. We will have a look at just one: the **Newton-Raphson algorithm** for locating the roots of a function. In this case, the function is the square root.

It is important because many versions of this algorithm rely on the explicit state managed via loops. Indeed, the *Hughes* paper provides a snippet of the Fortran code that emphasizes stateful, imperative processing.

The backbone of this approximation is the calculation of the next approximation from the current approximation. The `next_()` function takes `x`, an approximation to the `sqrt(n)` method and calculates a next value that brackets the proper root. Take a look at the following example:

```
In [183]: def next_(n, x):
              return (x+n/x)/2
```

This function computes a series of values $a_{i+1} = (a_i + n/a_i)/2$. The distance between the values is halved each time, so it will quickly get to converge on the value such that $a = n/a$ which means $a = \sqrt{n}$.

We do not want to call the method `next()` because this name would collide with a built-in function. We call it the `next_()` method so that we can follow the original presentation as closely as possible.

Here it is how the function looks when used it directly:

```
In [184]: n = 2
          f = lambda x: next_(n, x)
          a0 = 1.0
          [x for x in (a0, f(a0), f(f(a0)), f(f(f(a0))),)]
```

```
Out[184]: [1.0, 1.5, 1.4166666666666665, 1.4142156862745097]
```

We have defined the `f()` method as a `lambda` that will converge on $\sqrt{2}$.

We started with `1.0` as the initial value for a_0. Then we evaluated a sequence of recursive evaluations: $a_1 = f(a_0)$, $a_2 = f(f(a_0))$ and so on. We evaluated these functions using a generator expression so that we could round off each value. This makes the output easier to read and easier to use with `doctest`. The sequence appears to converge rapidly on $\sqrt{2}$.

We can write a function, which will (in principle) generate an infinite sequence of a_i values converging on the proper square root:

```
In [185]: def repeat(f, a):
              yield a
              for v in repeat(f, f(a)):
                  yield v
```

This function will generate approximations using a function, `f()`, and an initial value, `a`. If we provide the `next_()` function defined earlier, we can get a sequence of approximations to the square root of the `n` argument.

We have two ways to return all the values instead of returning a generator expression, which are as follows:

- We can write an explicit for loop as follows: `for x in some_iter: yield x`
- We can use the `yield from` statement as follows: `yield from some_iter`

Both techniques of yielding the values of a recursive generator function are equivalent. We will try to emphasize `yield from`. In some cases, however, the `yield` with a complex expression will be more clear than the equivalent mapping or generator expression.

Of course, we do not want the entire infinite sequence. We will stop generating values when two values are so close to each other that we can call either one the square root we are looking for. The common symbol for the value, which is close enough, is the Greek letter *epsilon*, ϵ, which can be thought of as the largest error we will tolerate.

In Python, we will have to be a little clever about taking items from an infinite sequence one at a time. It works out well to use a simple interface function that wraps a slightly more complex recursion. Take a look at the following code cell:

```
In [186]: def within(epsilon, iterable):
              def head_tail(epsilon, a, iterable):
                  b = next(iterable)
                  if abs(a - b) <= epsilon: return b
                  return head_tail(epsilon, b, iterable)
              return head_tail(epsilon, next(iterable), iterable)
```

We have defined an internal function, `head_tail()`, which accepts the tolerance, epsilon ϵ, an item from the iterable sequence, `a`, and the rest of the iterable sequence, `iterable`.

The next item from the `iterable` bound to a name `b`. If $\mid a - b \mid \leq \epsilon$, then the two values that are close enough together that we have found the square root. Otherwise, we use the `b` value in a recursive invocation of the `head_tail()` function to examine the next pair of values.

Our `within()` function merely seeks to properly initialize the internal `head_tail()` function with the first value from the iterable parameter.

Some functional programming languages offer a technique that will put a value back into an iterable sequence. In Python, this might be a kind of `unget()` or `previous()` method that pushes a value back into the iterator. Python iterables do not offer this kind of rich functionality.

We can use the three functions `next_()`, `repeat()`, and `within()` to create a square root function, as follows:

```
In [187]: def sqrt(a0, epsilon, n):
              return within(epsilon, repeat(lambda x: next_(n,x), a0))
```

```
In [188]: sqrt(1.,0.0001,2)
```

```
Out[188]: 1.4142135623746899
```

We have used the `repeat()` function to generate a (potentially) infinite sequence of values based on the `next_(n,x)` function. Our `within()` function will stop generating values in the sequence when it locates two values with a difference less than epsilon, .

When we use this version of the `sqrt()` method, we need to provide an initial seed value, `a0`, and an ϵ value. An expression like `sqrt(1.0, .0001, 2)` will start with an approximation of `1.0` and compute the value of $\sqrt{2}$ to within `0.0001`. For most applications, the initial `a0` value can be `1.0`. However, the closer it is to the actual square root, the more rapidly this method converges.

The original example of this approximation algorithm was shown in the *Miranda* language. It is not difficult to see that there are few profound differences between *Miranda* and *Python*. The biggest difference is Miranda's ability to construct `cons`, a value back into an `iterable`, doing a kind of `unget`. This parallelism between Miranda and Python gives us confidence that many kinds of functional programming can be easily done in Python.

CHAPTER

SEVEN

READING AND WRITING DATA

- Text files
- Structured Text Files
  - CSV files
  - JSON
  - HTML

7.1 Text Files

```
In [189]: text = "When you are courting a nice girl, an hour seems like a second.\n"
          text += "When you sit on a red-hot cinder, a second seems like an hour.\n"
          text += "That’s relativity."

In [190]: albert.txt
```

```
        ---------------------------------------------------------------------------

        NameError                                 Traceback (most recent call last)

        <ipython-input-190-11767518c1d6> in <module>()
    ----> 1 albert.txt


        NameError: name 'albert' is not defined
```

```
In [191]: # FIRST: Create an outfile object
          outfile = open('albert.txt','w')

          # SECOND: Write the text data
          outfile.write(text)

          # THIRD: Close the outfile object
          outfile.close()

In [192]: infile = open('albert.txt','r')
          text = infile.read()
          infile.close()

          print text
```

```
When you are courting a nice girl, an hour seems like a second.
When you sit on a red-hot cinder, a second seems like an hour.
That's relativity.
```

```
In [193]: with open('albert.txt','r') as infile:
              text = infile.read()
```

```
In [194]: print len(text)

145
```

7.2 Structured Text Data

- CSV: comma-separated values
- JSON: JavaScript Object Notation
- XML: EXtensible Markup Language
- HTML

```
In [195]: import csv
          with open('numbers.csv','w') as outfile:
              writer = csv.writer(outfile)
              writer.writerow(('first','second','third'))
              writer.writerow((1,3,6,8)) #oops
              writer.writerow((2,4,6))


          with open('numbers.csv','r') as infile:
              reader = csv.reader(infile)
              for lines in reader:
                  print lines

['first', 'second', 'third']
['1', '3', '6', '8']
['2', '4', '6']
```

JSON is a more flexible fromat than CSV for structured text

```
In [196]: data = {'name': 'Guido van Rossum',
                  'language': 'Python',
                  'similar': ['c','Modula-3','lisp'],
                  'users': 1000000}

          import json
          json.dumps(data)

Out[196]: '{"similar": ["c", "Modula-3", "lisp"], "name": "Guido van Rossum", "language": "Pyth

In [197]: with open('data.json','w') as outfile:
              outfile.write(json.dumps(data))

In [198]: with open('data.json','r') as infile:
              text = infile.read()
              data = json.loads(text)


          print data
          print data['name']

{u'similar': [u'c', u'Modula-3', u'lisp'], u'name': u'Guido van Rossum', u'language': u'Python'
Guido van Rossum
```

7.3 HTML: webAPIs with urllib2

```
In [199]: import urllib2
          response = urllib2.urlopen('http://vimeo.com/api/v2/video/57733101.json')
          data = json.load(response)[0]
```

```
In [200]: print data
          print '-'*100
          print data['title']
          print data['url']
          print data['duration']
          print data['stats_number_of_plays']
```

```
{u'upload_date': u'2013-01-19 04:01:15', u'height': 360, u'duration': 143, u'id': 57733101, u'u
---------------------------------------------------------------------------------------------------
The Good Man trailer
http://vimeo.com/57733101
143
4659
```

CHAPTER

EIGHT

PLOTTING WITH MATPLOTLIB

First rule in statistical modelling, plot your data if you can. Python has a great plotting library called Matplotlib. The IPython notebook interface we are using for these notes has that functionality built in.

Matplotlib is an excellent 2D and 3D graphics library for generating scientific figures. Some of the many advantages of this library includes:

- Easy to get started
- Support for $\LaTeX$ formatted labels and texts
- Great control of every element in a figure, including figure size and DPI.
- High-quality output in many formats, including PNG, PDF, SVG, EPS.
- GUI for interactively exploring figures *and* support for headless generation of figure files (useful for batch jobs).

The `matplotlib.pyplot` module contains the API for functions, such as `plot`. The convention is to import it as `plt`:

```
import matplotlib.pyplot as plt
```

You can launch IPython with `matplotlib` integration by typing:

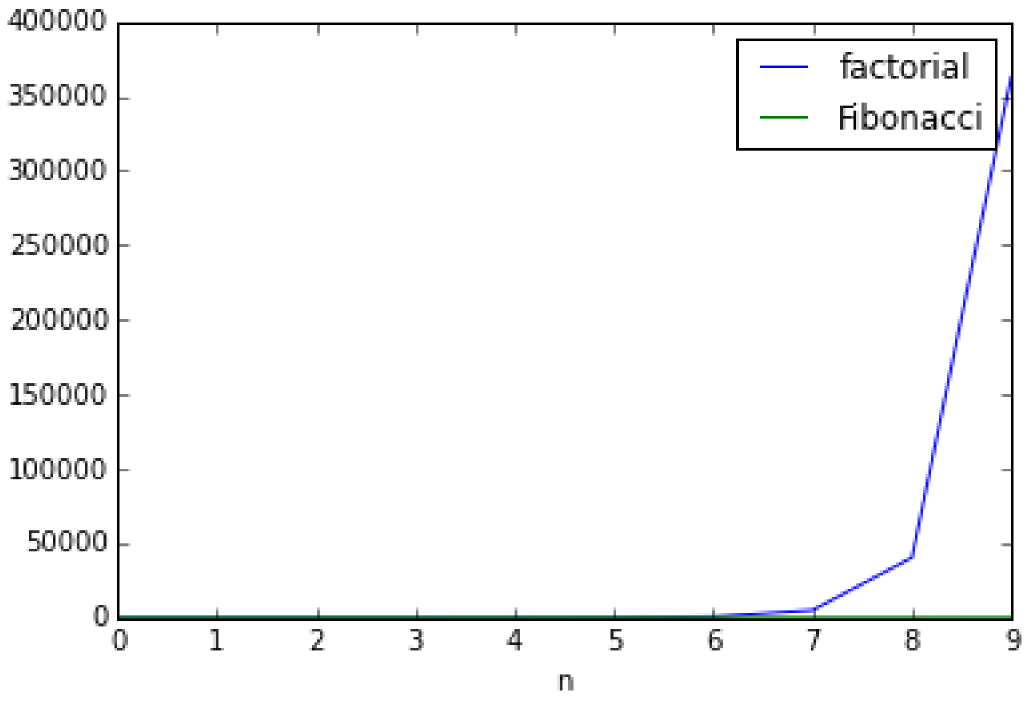
```
ipython --pylab

ipython notebook --pylab=inline
```

These two methods are the best ways to use matplotlib interactively. We you do this, you will have access to several `NumPy` and `pyplot` in the same namespace.

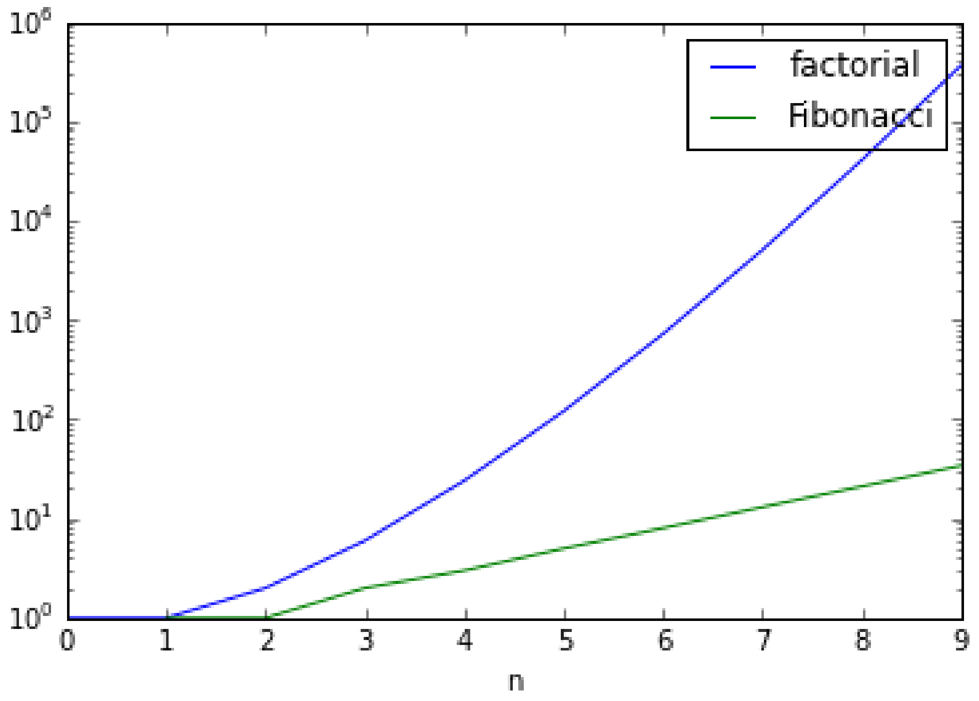
As an example, we have looked at two different functions, the Fibonacci function, and the factorial function, both of which grow faster than polynomially. Which one grows the fastest? Let's plot them.

```
In [201]: fibs = fibonacci(10)
          facts = []
          for i in range(10):
              facts.append(fact(i))

          # figsize(8,6)
          plt.plot(facts,label="factorial")
          plt.plot(fibs,label="Fibonacci")
          plt.xlabel("n")
          plt.legend()
```

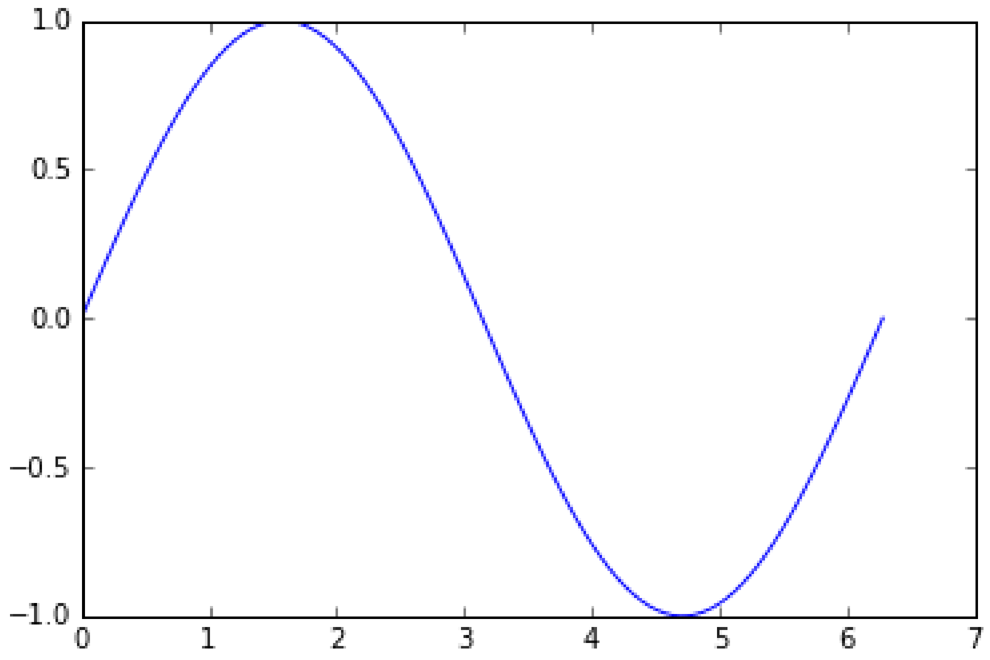
```
Out[201]: <matplotlib.legend.Legend at 0x1268e7a90>
```



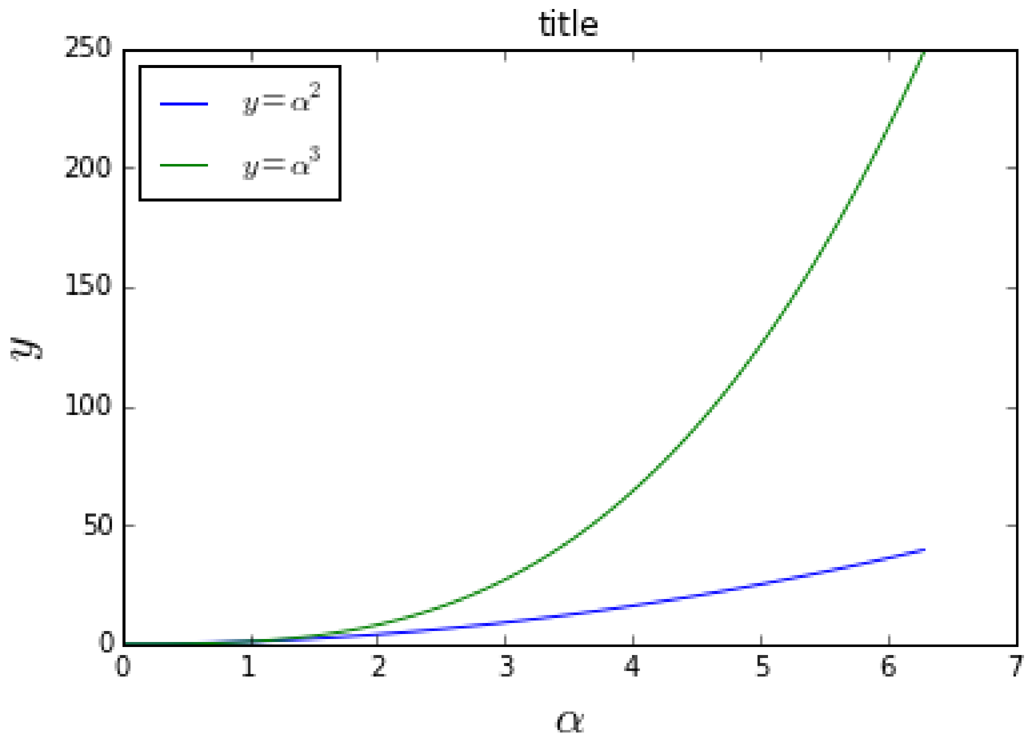
```
In [202]: # let us plot it on a semilog plot and we can see both more clearly

          plt.semilogy(facts,label="factorial")
          plt.semilogy(fibs,label="Fibonacci")
          plt.xlabel("n")
          plt.legend();
```



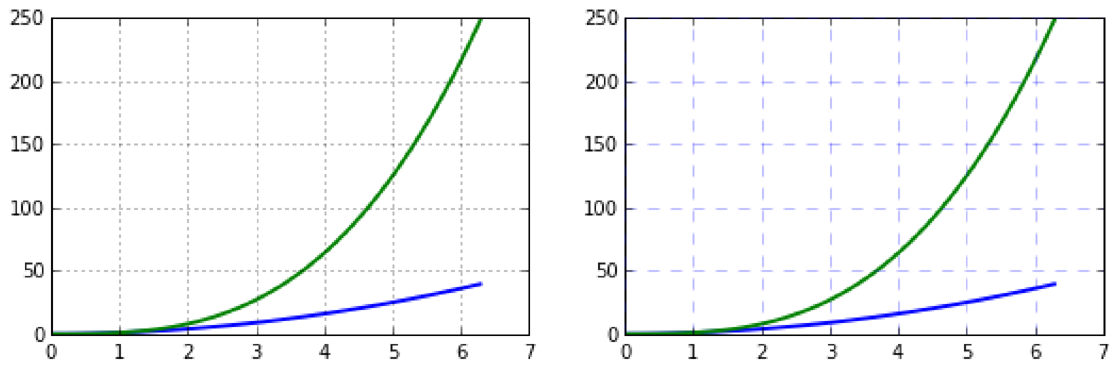
```
In [203]: x = np.linspace(0, 2*pi,100)
          y = np.sin(x)
          plt.plot (x,y);
```



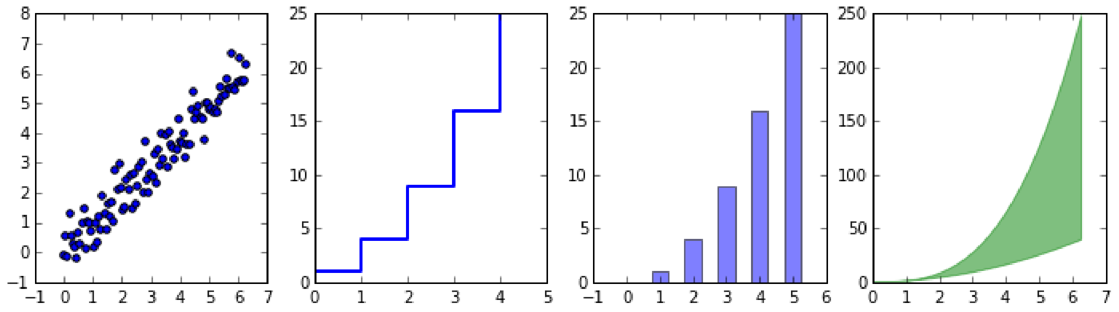
```
In [204]: fig, ax = plt.subplots();

          ax.plot(x, x**2, label=r"$y = \alpha^2$")
          ax.plot(x, x**3, label=r"$y = \alpha^3$")
          ax.set_xlabel(r'$\alpha$', fontsize=18)
          ax.set_ylabel(r'$y$', fontsize=18)
          ax.set_title('title')
          ax.legend(loc=2) # upper left corner

Out[204]: <matplotlib.legend.Legend at 0x10c4a2f50>
```



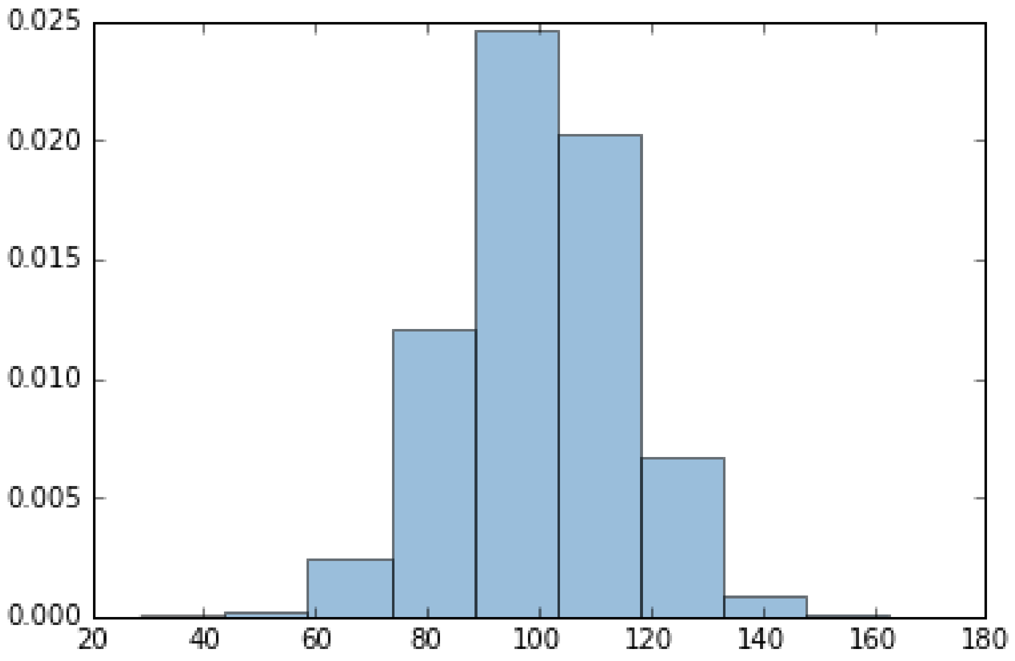
```
In [205]: fig, axes = plt.subplots(1, 2, figsize=(10,3))

          # default grid appearance
          axes[0].plot(x, x**2, x, x**3, lw=2)
          axes[0].grid(True)

          # custom grid appearance
          axes[1].plot(x, x**2, x, x**3, lw=2)
          axes[1].grid(color='b', alpha=0.5, linestyle='dashed', linewidth=0.5)
```



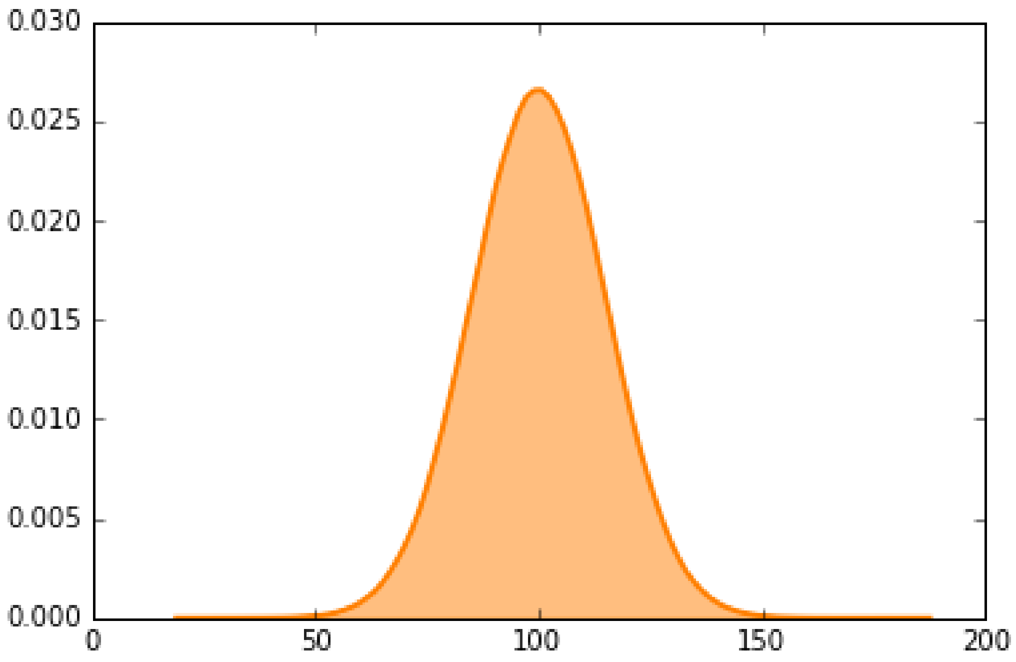
```
In [206]: n = np.array([0,1,2,3,4,5])
          fig, axes = plt.subplots(1, 4, figsize=(12,3))
          axes[0].scatter(x, x +0.5*np.random.randn(len(x)))
          axes[1].step(n, n**2, lw=2)
          axes[2].bar(n, n**2, align="center", width=0.5, alpha=0.5)
          axes[3].fill_between(x, x**2, x**3, color="green", alpha=0.5);
```



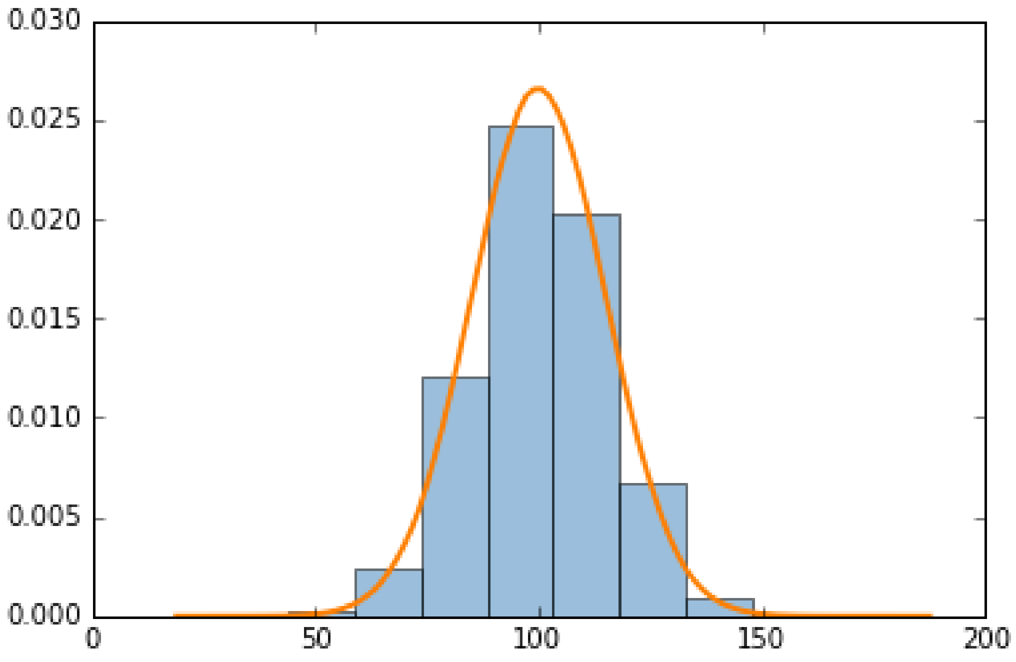
```
In [207]: # Histograms

          mu = 100
          sigma = 15
          x = mu + sigma * np.random.randn(3000000)

          plt.hist(x, 10, normed=1, alpha=0.5, facecolor='#377EB8')
          plt.show()
```



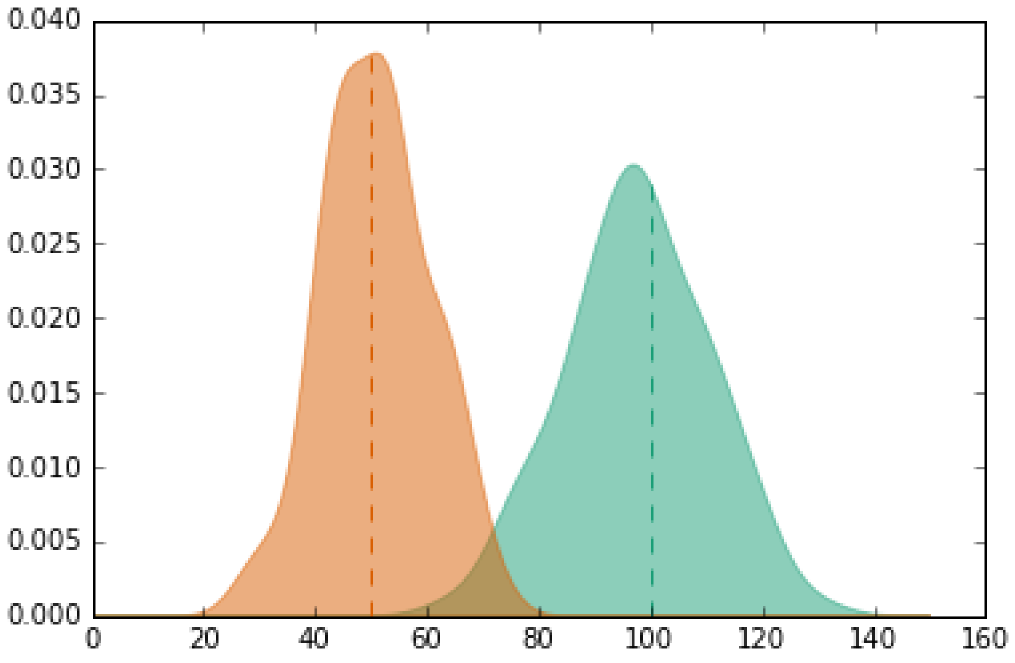
```
In [208]: density = stats.kde.gaussian_kde(x)
          xd = np.linspace(min(x)-10, max(x)+10, 100)
          plt.plot(xd, density(xd), lw=2, color='#FF7F00') #line
          plt.fill_between(xd, 0, density(xd), alpha=0.5, color='#FF7F00')
          plt.show()
```



```
In [209]: plt.hist(x, 10, normed=1, facecolor='#377EB8', alpha=0.5)
          plt.plot(xd, density(xd), lw=2, color='#FF7F00')
          plt.show()
```



```
In [210]: from SciPy import stats
          x1 = 100 + 15 * np.random.randn(100)
          x2 = 50 + 10 * np.random.randn(100)

          density1 = stats.kde.gaussian_kde(x1)
          density2 = stats.kde.gaussian_kde(x2)

          x = np.linspace(0, 150, 200)

In [211]: plt.fill_between(x, 0, density1(x), alpha=0.5, color='#1B9E77')
          plt.fill_between(x, 0, density2(x), alpha=0.5, color='#D95F02')

          plt.plot([100,100],[0,density1(100)], color ='#1B9E77', linewidth=1, linestyle="--")
          plt.plot([50,50],[0,density2(50)], color ='#D95F02', linewidth=1, linestyle="--")
          plt.show()
```



CHAPTER

NINE

NUMPY AND SCIPY: COMPUTING WITH VECTORS AND ARRAYS

NumPy is the fundamental Python package for scientific computing. It adds the capabilities of N-dimensional arrays, element-by-element operations (broadcasting), core mathematical operations like linear algebra, and the ability to wrap C/C++/Fortran code. SciPy is a package that utilizes NumPy arrays and manipulations to take on standard problems that scientists and engineers commonly face: integration, determining a function's maxima or minima, finding eigenvectors for large sparse matrices, testing whether two distributions are the same, and much more.

Both contain modules written in C and Fortran so that they're as fast as possible. Together, they give Python roughly the same capability that the Matlab program offers, but with an increase of speed that use to be higher than x300. (In fact, if you're an experienced Matlab user, there a guide to NumPy for Matlab users just for you.)

9.1 Why using NumPy for working with vectors and matrices

Fundamental to both NumPy and SciPy is the ability to work with vectors and matrices. You can create vectors from lists using the **array** command. You can pass in a second argument to **array** that gives the numeric type. There are a number of types listed here that your matrix can be. Some of these are aliased to single character codes. The most common ones are 'd' (double precision floating point number), 'D' (double precision complex number), and 'i' (int32).

You can create NumPy arrays using the `NumPy.array` function. It takes a list-like object (or another array) as input and, optionally, a string expressing its data type. You can interactively test the array creation using an IPython shell as follows:

```
In [212]: a = np.array([1,2,3,4,5,6])
          a.dtype

Out[212]: dtype('int64')

In [213]: a = np.array([1, 2, 3], dtype='float32')
          a.astype('float32')

Out[213]: array([ 1.,  2.,  3.], dtype=float32)

In [214]: print np.array([1,2,3,4,5,6])
          print
          print 'Same array and different types'
          print np.array([1,2,3,4,5,6],'d')
          print np.array([1,2,3,4,5,6],'D')
          print np.array([1,2,3,4,5,6],'i')
          print
          print 'Double precision floating point MATRIX'
          print np.array([[0,1],[1,0]],'d')

[1 2 3 4 5 6]

Same array and different types
[ 1.  2.  3.  4.  5.  6.]
[ 1.+0.j  2.+0.j  3.+0.j  4.+0.j  5.+0.j  6.+0.j]
[1 2 3 4 5 6]
```

```
Double precision floating point MATRIX
[[ 0.  1.]
 [ 1.  0.]]
```

Python stores data in several different ways, but the most popular methods are `lists` and `dictionaries`. The Python `list` object can store nearly any type of Python object as an element. But operating on the elements in a list can only be done through iterative loops, which is computationally inefficient in Python. The NumPy package enables users to overcome the shortcomings of the Python lists by providing a data storage object called **ndarray**.

The `ndarray` is similar to lists, but rather than being highly flexible by storing different types of objects in one list, only the same type of element can be stored in each column. For example, with a Python list, you could make the first element a list and the second another list or dictionary. With NumPy arrays, you can only store the same type of element, e.g., all elements must be floats, integers, or strings. Despite this limitation, `ndarray` wins hands down when it comes to operation times, as the operations are sped up significantly. Using the `%timeit` magic command in IPython, we compare the power of NumPy ndarray versus Python lists in terms of speed.

```
In [215]: # Create an array with 10^7 elements.

          arr = np.arange(1e7)
          %timeit arr * 1.1

10 loops, best of 3: 25.5 ms per loop
```

```
In [216]: larr = arr.tolist()

          def list_times(alist, scalar):
              for i, val in enumerate(alist):
                  alist[i] = val * scalar
              return alist

          %timeit list_times(larr, 1.1)

1 loops, best of 3: 917 ms per loop
```

The `ndarray` operation is $\sim$ 36 faster than the Python loop in this example. Are you convinced that the NumPy ndarray is the way to go? From this point on, we will try to be working with the array objects instead of lists when possible.

9.2 Basic NumPy functionality for Matrices

```
In [217]: print np.zeros(3,'d')
          print
          print np.zeros((3,3),'d')

[ 0.  0.  0.]

[[ 0.  0.  0.]
 [ 0.  0.  0.]
 [ 0.  0.  0.]]
```

```
In [218]: print np.zeros((3,1),'d')
          print
          print np.zeros((1,3),'d')

[[ 0.]
 [ 0.]
 [ 0.]]

[[ 0.  0.  0.]]
```

```
In [219]: print np.identity(3,'d')
          print
          print np.ones((3,3),'d')
          print
          print np.identity(3,'d')*333

[[ 1.  0.  0.]
 [ 0.  1.  0.]
 [ 0.  0.  1.]]

[[ 1.  1.  1.]
 [ 1.  1.  1.]
 [ 1.  1.  1.]]

[[ 333.    0.    0.]
 [   0.  333.    0.]
 [   0.    0.  333.]]


In [220]: # Diagonal of square matrix

          np.diag([1,2,3,4,5])

Out[220]: array([[1, 0, 0, 0, 0],
                 [0, 2, 0, 0, 0],
                 [0, 0, 3, 0, 0],
                 [0, 0, 0, 4, 0],
                 [0, 0, 0, 0, 5]])

In [221]: # Indexing and Slicing

          a = np.identity(3,'d')*333
          print a
          print

          print 'Indexes of elements bigger than 0:'
          index = np.where(a > 2)
          print index
          print

          print 'Actual values bigger than 0:'
          print a[index]
          print

          index = arr > 2
          print(index)

[[ 333.    0.    0.]
 [   0.  333.    0.]
 [   0.    0.  333.]]

Indexes of elements bigger than 0:
(array([0, 1, 2]), array([0, 1, 2]))

Actual values bigger than 0:
[ 333.  333.  333.]

[False False False ...,  True  True  True]
```

9.3 Linear Algebra with NumPy

NumPy arrays do not behave like matrices in linear algebra by default. Instead, the operations are mapped from each element in one array onto the next. This is quite a useful feature, as loop operations can be done away with for efficiency.

```
In [222]: np.identity(2,'d') + np.array([[1,1],[1,2]])

Out[222]: array([[ 2.,  1.],
                 [ 1.,  3.]])

In [223]: # Elementwise multiplication or broadcasting

          print np.identity(2)*np.ones((2,2))
          print

          # Matrix multiplication

          print np.dot(np.identity(2),np.ones((2,2)))

[[ 1.  0.]
 [ 0.  1.]]

[[ 1.  1.]
 [ 1.  1.]]
```

But what about when transposing or a dot multiplication are needed? Without invoking other classes, you can use the built-in `NumPy.dot` and `NumPy.transpose` to do such operations.

```
In [224]: m = np.array([[1,2],[3,4]])

          print 'Original matrix m: \n',m
          print

          # Transpose matrices

          print 'Transpose matrix m: \n',m.transpose()
          print

          # Inverting matrices

          print 'Inverted matrix m: \n', np.linalg.inv(m)
          print

          # Determinant of m

          print 'Determinant of m: \n', np.linalg.det(m)
          print

          # Eigen Values

          print 'Eigen values: \n', np.linalg.eigvals(m)

Original matrix m:
[[1 2]
 [3 4]]

Transpose matrix m:
[[1 3]
 [2 4]]

Inverted matrix m:
[[-2.   1. ]
 [ 1.5 -0.5]]

Determinant of m:
-2.0

Eigen values:
[-0.37228132  5.37228132]
```

Or the math purist can use the `NumPy.matrix` object instead.

```
In [225]: # Solving a matrix systyem using directly matrices type:
```

```
           A = np.matrix([[3, 6, -5], [1, -3, 2],[5, -1, 4]])
           B = np.matrix([[12], [-2], [10]])

           # Solving for the variables, where we invert A

           X = A ** (-1) * B

           print(X)
[[ 1.75]
 [ 1.75]
 [ 0.75]]


In [226]: # Solving the same system of linear equations using the solve command and np.arrays

           A = np.array([[3, 6, -5], [1, -3, 2],[5, -1, 4]])
           b = np.array([12, -2, 10])

           print 'Solution using np.arrays:'
           x = np.linalg.inv(a).dot(b)
           print(x)
           print
           print 'Solution using np.linalg.solve():'
           print np.linalg.solve(A,b)
           print
           print 'Solution using np.linalg.lstsq():'
           print np.linalg.lstsq(A,b)
Solution using np.arrays:
[ 0.03603604 -0.00600601  0.03003003]

Solution using np.linalg.solve():
[ 1.75  1.75  0.75]

Solution using np.linalg.lstsq():
(array([ 1.75,  1.75,  0.75]), array([], dtype=float64), 3, array([ 9.25213364,  6.25912522,  1
```

Both methods of approaching linear algebra operations are viable, but which one is the best? The `NumPy.matrix` method is syntactically the simplest. However, `NumPy.array` is the most practical. First, the NumPy array is the standard for using nearly anything in the scientific Python environment, so bugs pertaining to the linear algebra operations will be less frequent than with `NumPy.matrix` operations. Furthermore, in examples such as the two shown above, the `NumPy.array` method is computationally faster.

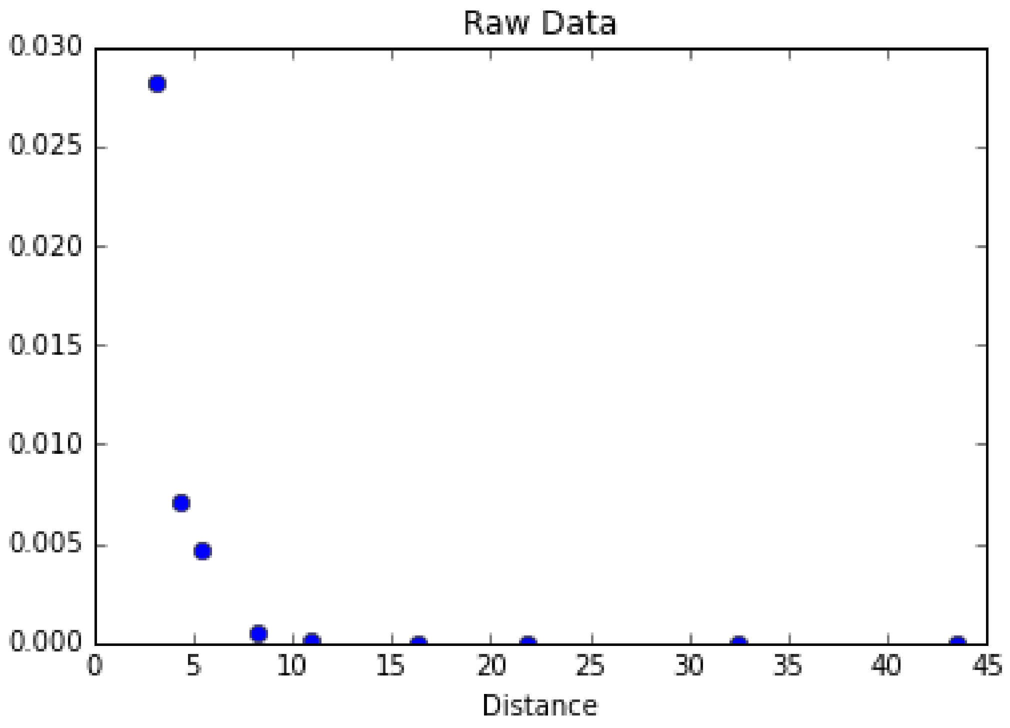
Passing data structures from one class to another can become cumbersome and lead to unexpected results when not done correctly. This would likely happen if one were to use `NumPy.matrix` and then pass it to `NumPy.array` for further operations. Sticking with one data structure will lead to fewer headaches and less worry than switching between matrices and arrays. It is advisable, then, to use `NumPy.array` whenever possible.

9.4 Code Example: Least-squares fitting

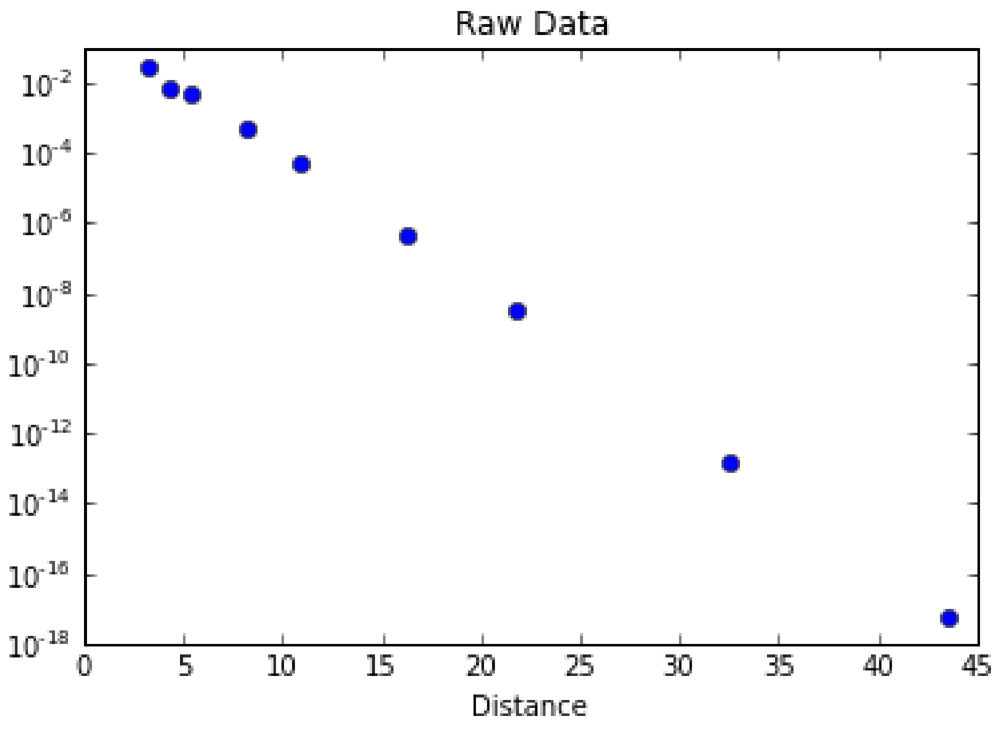
Very often we deal with some data that we want to fit to some sort of expected behaviour. Say we have the following:

```
In [227]: raw_data = """\
           3.1905781584582433,0.028208609537968457
           4.346895074946466,0.007160804747670053
           5.374732334047101,0.0046962988461934805
           8.201284796573875,0.0004614473299618756
           10.899357601713055,0.00005038370219939726
           16.295503211991434,4.377451812785309e-7
           21.82012847965739,3.0799922117601088e-9
           32.48394004282656,1.524776208284536e-13
           43.53319057815846,5.5012073588707224e-18"""
```

```
          type(raw_data)

Out[227]: str

In [228]: # DATA WRANGLING: Let us work a little with raw_data to make it mathematically accept

          data = []
          for line in raw_data.splitlines():
              words = line.split(',')
              data.append(map(float,words))
          data = np.array(data)
          print data

[[  3.19057816e+00   2.82086095e-02]
 [  4.34689507e+00   7.16080475e-03]
 [  5.37473233e+00   4.69629885e-03]
 [  8.20128480e+00   4.61447330e-04]
 [  1.08993576e+01   5.03837022e-05]
 [  1.62955032e+01   4.37745181e-07]
 [  2.18201285e+01   3.07999221e-09]
 [  3.24839400e+01   1.52477621e-13]
 [  4.35331906e+01   5.50120736e-18]]

In [229]: # DATA VISUALIZATION: How these data looks like...

          plt.title("Raw Data")
          plt.xlabel("Distance")
          plt.plot(data[:,0],data[:,1],'bo');
```



```
In [230]: plt.title("Raw Data")
          plt.xlabel("Distance")
          plt.semilogy(data[:,0],data[:,1],'bo');
```



For a pure exponential decay like this, we can fit the log of the data to a straight line. The above plot suggests this is a good approximation. Given a function

$$y = Ae^{-ax}$$

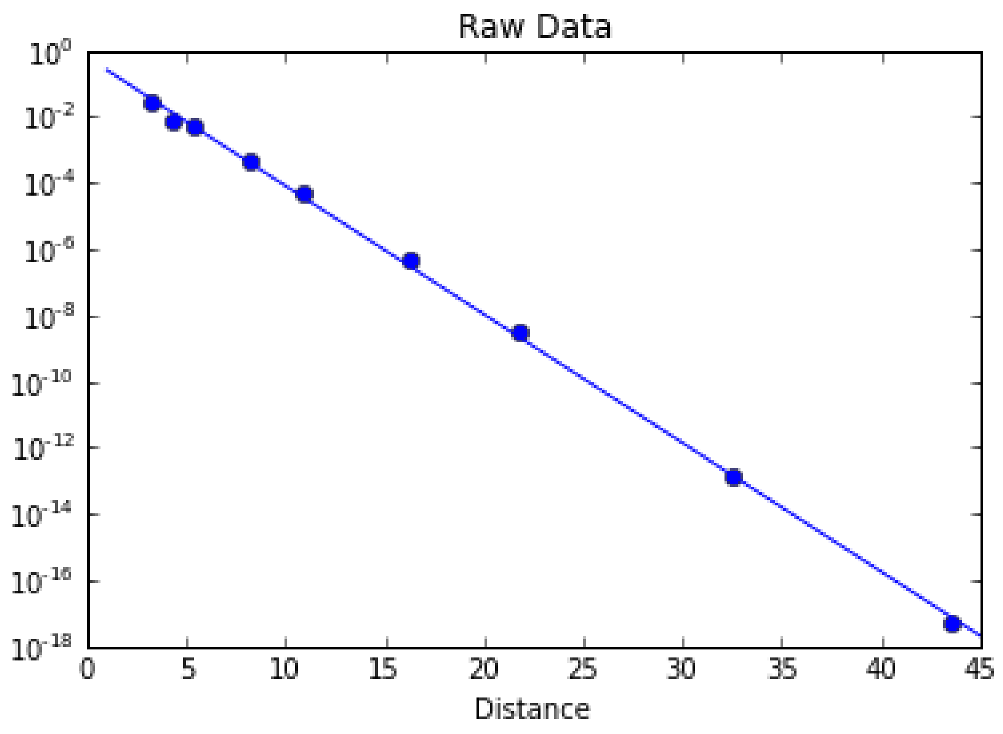
$$\log(y) = \log(A) - ax$$

Thus, if we fit the log of the data versus x, we should get a straight line with slope a, and an intercept that gives the constant A.

There's a NumPy function called **polyfit** that will fit data to a polynomial form. We'll use this to fit to a straight line (a polynomial of order 1)

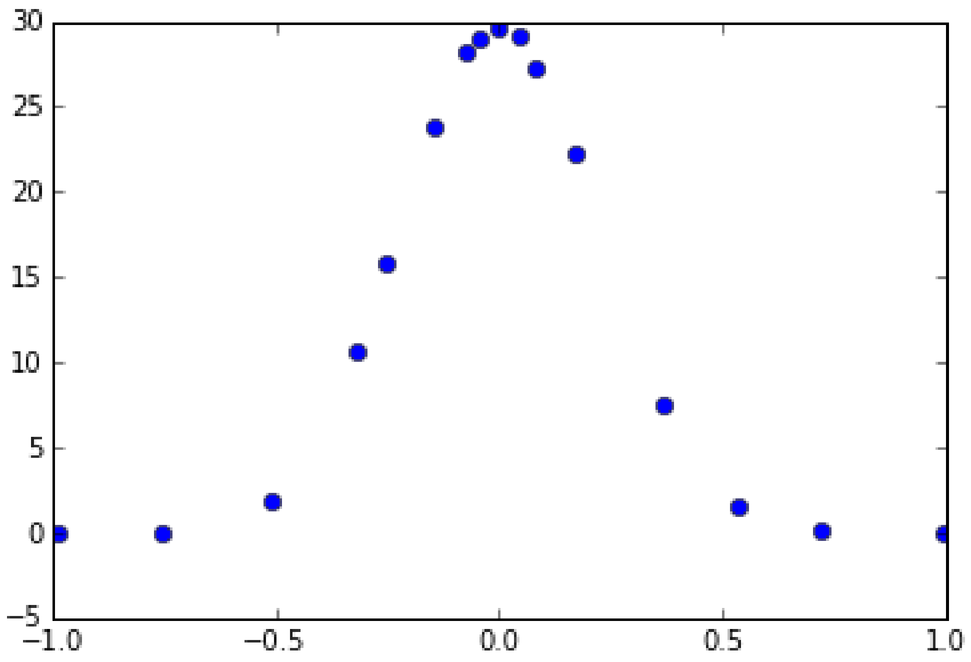
```
In [231]: params = np.polyfit(data[:,0],np.log(data[:,1]),1)
          a = params[0]
          A = np.exp(params[1])
```

```
In [232]: x = np.linspace(1,45)
          plt.title("Raw Data")
          plt.xlabel("Distance")
          plt.semilogy(data[:,0],data[:,1],'bo')
          plt.semilogy(x,A*np.exp(a*x),'b-');
```



Let us consider another example a little bit more complicate:

```
In [233]: gauss_data = """\
          -0.9902286902286903,1.4065274110372852e-19
          -0.7566104566104566,2.2504438576596563e-18
          -0.5117810117810118,1.9459459459459454
          -0.31887271887271884,10.621621621621626
          -0.250997150997151,15.891891891891893
          -0.1463309463309464,23.756756756756754
          -0.07267267267267263,28.135135135135133
          -0.04426734426734419,29.02702702702703
          -0.0015939015939017698,29.675675675675677
          0.04689304689304685,29.10810810810811
          0.0840994840994842,27.324324324324326
          0.1700546700546699,22.216216216216214
          0.370878570878571,7.540540540540545
          0.5338338338338338,1.621621621621618
          0.722014322014322,0.08108108108108068
          0.9926849926849926,-0.08108108108108646"""

          data = []
          for line in gauss_data.splitlines():
              words = line.split(',')
              data.append(map(float,words))
          data = np.array(data)

          plt.plot(data[:,0],data[:,1],'bo');
```

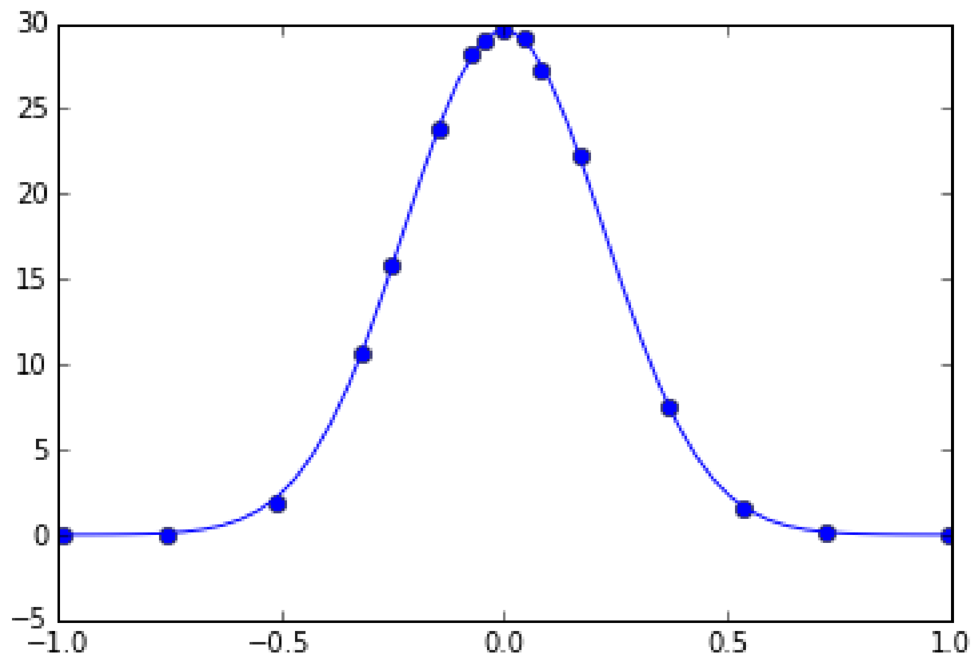


let's use the **curve_fit** function from SciPy, which can fit to arbitrary functions. You can learn more using help(curve_fit).

First define a general Gaussian function to fit to.

```
In [234]: def gauss(x,A,a): return A*np.exp(a*x**2)
```

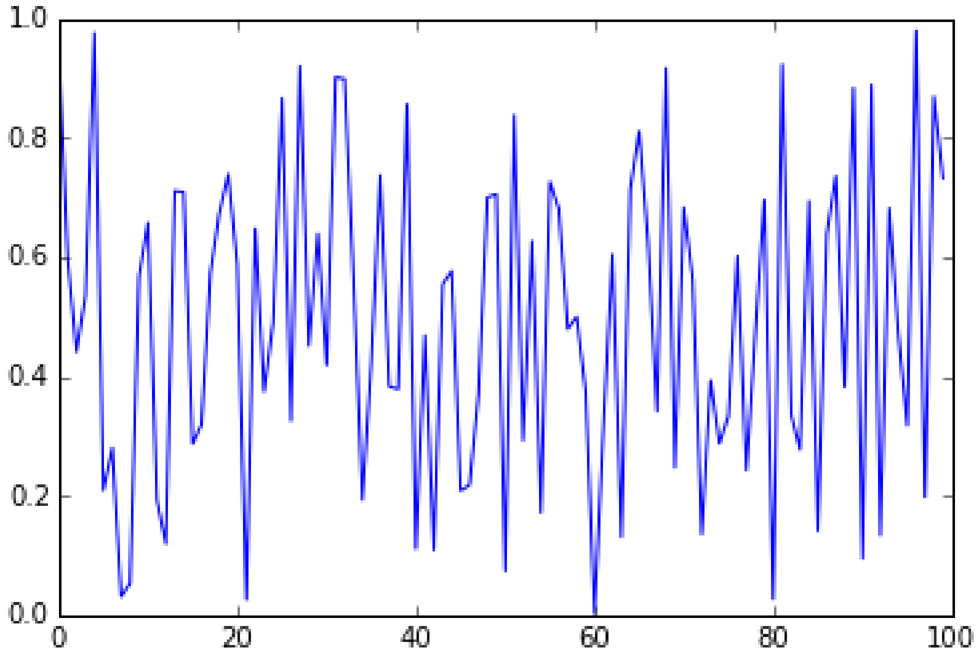
```
In [235]: from SciPy.optimize import curve_fit

          params,conv = curve_fit(gauss,data[:,0],data[:,1])
          x = np.linspace(-1,1)
          plt.plot(data[:,0],data[:,1],'bo');
          A,a = params
          plt.plot(x,gauss(x,A,a),'b-');
```



9.5 Code Example: Monte-Carlo simulation and π computation

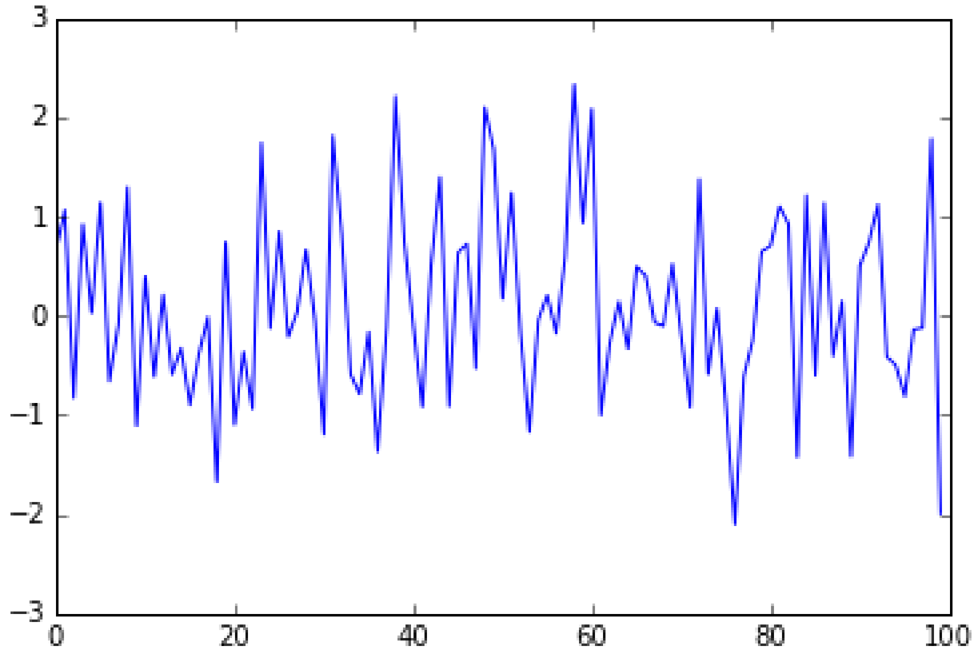
Python has good random number generators in the standard library. The **random()** function gives pseudorandom numbers uniformly distributed between 0 and 1:

```
In [236]: from random import random
          rands = []
          for i in range(100):
              rands.append(random())
          plt.plot(rands);
```



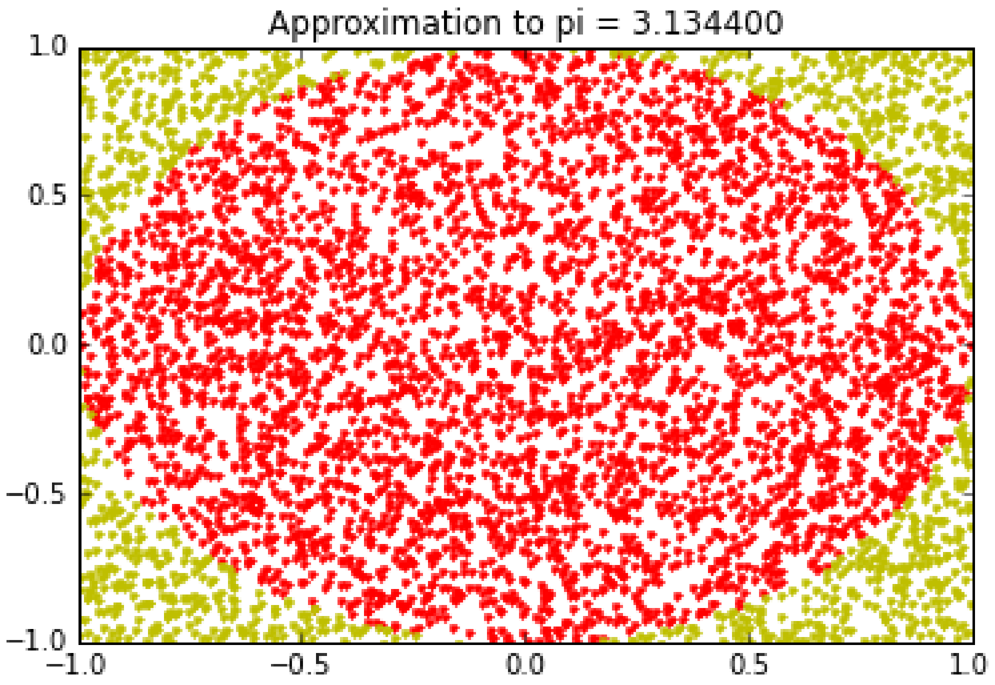
random() uses the Mersenne Twister algorithm, which is a highly regarded pseudorandom number generator. There are also functions to generate random integers, to randomly shuffle a list, and functions to pick random numbers from a particular distribution, like the normal distribution:

```
In [237]: from random import gauss
          grands = []
          for i in range(100):
              grands.append(gauss(0,1))
          plt.plot(grands);
```



Here you have an interesting program to compute π by taking random numbers as x and y coordinates, and counting how many of them were in the unit circle. For example:

```
In [238]: npts = 5000
          xs = 2*np.random.rand(npts)-1
          ys = 2*np.random.rand(npts)-1
          r = xs**2+ys**2
          ninside = (r<1).sum()
          # figsize(6,6) # make the figure square
          plt.title("Approximation to pi = %f" % (4*ninside/float(npts)));
          plt.plot(xs[r<1],ys[r<1],'r.');
          plt.plot(xs[r>1],ys[r>1],'y.');
          #figsize(8,6) # change the figsize back to 4x3 for the rest of the notebook

Out[238]: [<matplotlib.lines.Line2D at 0x1358b7c90>]
```



9.6 Code Example: High-Performance Simulation with Cython

In this example, we aim to show how to integrate NumPy arrays and Cython code. We will also see how calls to Python functions inside tight loops can be optimized by converting the Python functions into C functions.

Here, we simulate an sample of a stochastic process simulation, namely a **Brownian motion**. This process describes the trajectory of a particle starting at `x=0`, and making random steps of `+dx` or `-dx` at each discrete time step (also called *random walk*), with `dx` being a small constant. This type of process appears frequently in finance, economy, physics, biology, and so on.

This specific process can be simulated very efficiently using **NumPy** methods on `ndarrays` as `cumsum()` and `rand()` functions. However, more complex processes may need to be simulated, for example, some models require instantaneous jumps when the position reaches a threshold. In these cases, vectorization is not an option and a manual loop could be unavoidable.

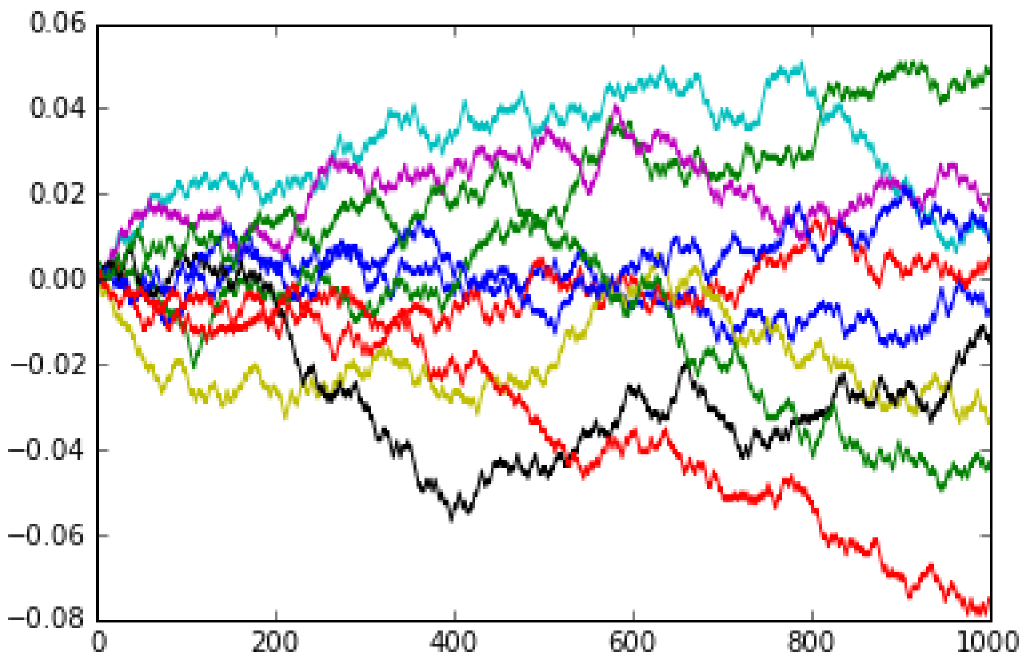
```
In [31]: import numpy as np

         def step():
             return np.sign(np.random.random(1)-.5)
```

The step function returns a random +1 or −1 value. It uses NumPy's `sign()` and `rand()` functions.

Below, in the `sim1()` function, the trajectory is first initialized as a **NumPy** vector with zeros. Then, at each iteration, a new random step is added to the trajectory. The then function returns the full trajectory. The following is an example of a trajectory:

```
In [33]: def sim1(n):
             x = np.zeros(n)
             dx = 1./n
             for i in xrange(n-1):
                 x[i+1] = x[i] + dx * step()
             return x
```

```
In [57]: for __ in range(10): plt.plot(sim1(1000));
```



```
In [37]: n = 1000
         %timeit sim1(n)
```

```
100 loops, best of 3: 4.65 ms per loop
```

For the **Cython** version, we will do two things. *First*, we will add `C` types for all local variables as well as for the NumPy array containing the trajectory. Also, we will convert the `step()` function to a pure `C` function that does not call any **NumPy** function. We will rather call pure `C` functions that are defined in the `C` standard library.

```
In [47]: %load_ext cython
```

```
The cython extension is already loaded. To reload it, use:
  %reload_ext cython
```

```
In [48]: %%cython

         import numpy as np
         cimport numpy as np
         DTYPE = np.double
         ctypedef np.double_t DTYPE_t

         # We redefine step() as a pure C function, using only
         # the C standard library.

         from libc.stdlib cimport rand, RAND_MAX
         from libc.math cimport round

         cdef double step():
             return 2 * round(float(rand()) / RAND_MAX) - 1

         def sim2(int n):
             # Local variables should be defined as C variables.
             cdef int i
             cdef double dx = 1. / n
             cdef np.ndarray[DTYPE_t, ndim=1] x = np.zeros(n, dtype=DTYPE)
             for i in range(n - 1):
                 x[i+1] = x[i] + dx * step()
             return x
```

We first need to import the standard **NumPy** library as well as a special `C` library, also called **NumPy**, which is part of the **Cython package**, with `cimport`. We define the NumPy `dtype double` and the corresponding `C dtype double_t` with `ctypedef`. It allows to define the exact type of the `x` array at compile-time rather than execution-time, resulting in major speed improvements. The number of dimensions of `x` is also specified inside the `sim2()` function. All local variables are defined as C variables with C types.

The `step()` function has been entirely rewritten. It is now a pure `C` function (defined with `cdef`). It uses the `rand()` function of the `C` standard library, which returns a random number between `0` and `RAND_MAX`. The `round()` function of the math library is also used to generate a random +1 or -1 value.

```
In [49]: %timeit sim2(n)

100000 loops, best of 3: 11.2 µs per loop
```

The **Cython** version is more than 400 times faster than the **Python** version. The main reason for this dramatic speed improvement is that the **Cython** version uses only pure **C** code. All variables are C variables, and the calls to step, which previously required costly calls to a **Python** function, now only involve calls to a pure **C** function, thereby reducing considerably the Python overhead inside the loop.

```
In [46]: %%cython
         cimport cython
         from libc.math cimport exp, sqrt, pow, log, erf

         @cython.cdivision(True)
         cdef double std_norm_cdf(double x) nogil:
             return 0.5*(1+erf(x/sqrt(2.0)))

         @cython.cdivision(True)
         def black_scholes(double s, double k, double t, double v,
                          double rf, double div, double cp):
             """Price an option using the Black-Scholes model.

             s : initial stock price
             k : strike price
             t : expiration time
             v : volatility
             rf : risk-free rate
             div : dividend
             cp : +1/-1 for call/put
             """
             cdef double d1, d2, optprice
```

```
            with nogil:
                d1 = (log(s/k)+(rf-div+0.5*pow(v,2))*t)/(v*sqrt(t))
                d2 = d1 - v*sqrt(t)
                optprice = cp*s*exp(-div*t)*std_norm_cdf(cp*d1) - \
                    cp*k*exp(-rf*t)*std_norm_cdf(cp*d2)
            return optprice
```

```
In [52]: black_scholes(100.0, 100.0, 1.0, 0.3, 0.03, 0.0, -1)
```

```
Out[52]: 10.327861752731728
```

Apart from **Cython**, there are other packages that accelerate Python code.

- **SciPy.weave** (http://www.scipy.org/Weave) is a SciPy subpackage that allows the inclusion of C/C++ code within Python code.
- **Numba** (http://numba.pydata.org/) uses just-in-time LLVM compilation to accelerate a pure Python code considerably by compiling it dynamically and transparently. It integrates nicely with NumPy arrays. Its installation requires llvmpy and meta.
- **Theano** (http://deeplearning.net/software/theano/), which allows to define, optimize, and evaluate mathematical expressions on arrays very efficiently by compiling them transparently on the CPU or on the graphics card.
- **Numexpr** (https://code.google.com/p/numexpr/) can compile array expressions and take advantage of vectorized CPU instructions and multi-core processors.
- **Blaze** (http://blaze.pydata.org/) is a project that is still in early development at the time of writing, and aims at combining all these dynamic compilation technologies together into a unified framework. It will also extend the notion of multidimensional array by allowing type and shape heterogeneity, missing values, labeled dimensions (such as in Pandas), and so on. Being developed by the creators of NumPy, it is likely to be a central project in the Python computing community in the near future.
- Finally, **PyOpenCL** (http://mathema.tician.de/software/pyopencl) and **PyCUDA** (http://mathema.tician.de/software/pycuda) are Python wrappers to **OpenCL** and **CUDA**. These libraries implement C-like, low-level languages that can be compiled on modern graphics cards for taking advantage of their massively parallel architecture. Indeed, graphics cards contain hundreds of specialized cores that can process a function very efficiently on a large number of elements (Single Instruction Multiple Data (SIMD) paradigm). The speed improvement can be more than one order of magnitude faster compared to pure C code. **OpenCL** is an open standard language, whereas **CUDA** is a proprietary language owned by Nvidia Corporation. **CUDA** code runs on **Nvidia** cards only, whereas **OpenCL** is supported by most graphics cards as well as most CPUs. In the latter case, the same code is compiled on the CPU and takes advantage of multi-core processors and vectorized instructions.

CHAPTER TEN

FURTHER READING

- http://www.python.org - The official web page of the Python programming language.
- http://www.python.org/dev/peps/pep-0008 - Style guide for Python programming. Highly recommended.
- http://www.greenteapress.com/thinkpython/ - A free book on Python programming.
- Python Essential Reference - A good reference book on Python programming.