1 An Introduction to the Scientific Python Ecosystem

While the Python language is an excellent tool for general-purpose programming, with a highly readable syntax, rich and powerful data types (strings, lists, sets, dictionaries, arbitrary length integers, etc) and a very comprehensive standard library, it was not designed specifically for mathematical and scientific computing. Neither the language nor its standard library have facilities for the efficient representation of multidimensional datasets, tools for linear algebra and general matrix manipulations (an essential building block of virtually all technical computing), nor any data visualization facilities.

In particular, Python lists are very flexible containers that can be nested arbitrarily deep and which can hold any Python object in them, but they are poorly suited to represent efficiently common mathematical constructs like vectors and matrices. In contrast, much of our modern heritage of scientific computing has been built on top of libraries written in the Fortran language, which has native support for vectors and matrices as well as a library of mathematical functions that can efficiently operate on entire arrays at once.

1.1 Scientific Python: a collaboration of projects built by scientists

The scientific community has developed a set of related Python libraries that provide powerful array facilities, linear algebra, numerical algorithms, data visualization and more. In this appendix, we will briefly outline the tools most frequently used for this purpose, that make "Scientific Python" something far more powerful than the Python language alone.

For reasons of space, we can only describe in some detail the central Numpy library, but below we provide links to the websites of each project where you can read their documentation in more detail.

First, let's look at an overview of the basic tools that most scientists use in daily research with Python. The core of this ecosystem is composed of:

- Numpy: the basic library that most others depend on, it provides a powerful array type that can represent multidmensional datasets of many different kinds and that supports arithmetic operations. Numpy also provides a library of common mathematical functions, basic linear algebra, random number generation and Fast Fourier Transforms. Numpy can be found at numpy.scipy.org

- Scipy: a large collection of numerical algorithms that operate on numpy arrays and provide facilities for many common tasks in scientific computing, including dense and sparse linear algebra support, optimization, special functions, statistics, n-dimensional image processing, signal processing and more. Scipy can be found at scipy.org.

- Matplotlib: a data visualization library with a strong focus on producing high-quality output, it supports a variety of common scientific plot types in two and three dimensions, with precise control over the final output and format for publication-quality results. Matplotlib can also be controlled interactively allowing graphical manipulation of your data (zooming, panning, etc) and can be used with most modern user interface toolkits. It can be found at matplotlib.sf.net.

- IPython: while not strictly scientific in nature, IPython is the interactive environment in which many scientists spend their time. IPython provides a powerful Python shell that integrates tightly with Matplotlib and with easy access to the files and operating system, and which can

execute in a terminal or in a graphical Qt console. IPython also has a web-based notebook interface that can combine code with text, mathematical expressions, figures and multimedia. It can be found at ipython.org.

While each of these tools can be installed separately, in our opinion the most convenient way today of accessing them (especially on Windows and Mac computers) is to install the Free Edition of the Enthought Python Distribution which contain all the above. Other free alternatives on Windows (but not on Macs) are Python(x,y) and Christoph Gohlke's packages page.

These four 'core' libraries are in practice complemented by a number of other tools for more specialized work. We will briefly list here the ones that we think are the most commonly needed:

- Sympy: a symbolic manipulation tool that turns a Python session into a computer algebra system. It integrates with the IPython notebook, rendering results in properly typeset mathematical notation. sympy.org.
- Mayavi: sophisticated 3d data visualization; code.enthought.com/projects/mayavi.
- Cython: a bridge language between Python and C, useful both to optimize performance bottlenecks in Python and to access C libraries directly; cython.org.
- Pandas: high-performance data structures and data analysis tools, with powerful data alignment and structural manipulation capabilities; pandas.pydata.org.
- Statsmodels: statistical data exploration and model estimation; statsmodels.sourceforge.net.
- Scikit-learn: general purpose machine learning algorithms with a common interface; scikit-learn.org.
- Scikits-image: image processing toolbox; scikits-image.org.
- NetworkX: analysis of complex networks (in the graph theoretical sense); networkx.lanl.gov.
- PyTables: management of hierarchical datasets using the industry-standard HDF5 format; www.pytables.org.

Beyond these, for any specific problem you should look on the internet first, before starting to write code from scratch. There's a good chance that someone, somewhere, has written an open source library that you can use for part or all of your problem.

1.2 A note about the examples below

In all subsequent examples, you will see blocks of input code, followed by the results of the code if the code generated output. This output may include text, graphics and other result objects. These blocks of input can be pasted into your interactive IPython session or notebook for you to execute. In the print version of this document, a thin vertical bar on the left of the blocks of input and output shows which blocks go together.

If you are reading this text as an actual IPython notebook, you can press `Shift-Enter` or use the 'play' button on the toolbar (right-pointing triangle) to execute each block of code, known as a 'cell' in IPython:

```
# This is a block of code, below you'll see its output
print "Welcome to the world of scientific computing with Python!"
```

```
Welcome to the world of scientific computing with Python!
```

2 Motivation: the trapezoidal rule

In subsequent sections we'll provide a basic introduction to the nuts and bolts of the basic scientific python tools; but we'll first motivate it with a brief example that illustrates what you can do in a few lines with these tools. For this, we will use the simple problem of approximating a definite integral with the trapezoid rule:

$$\int_a^b f(x)\,dx \approx \frac{1}{2}\sum_{k=1}^{N}(x_k - x_{k-1})\,(f(x_k) + f(x_{k-1}))\,.$$

Our task will be to compute this formula for a function such as:

$$f(x) = (x-3)(x-5)(x-7) + 85$$

integrated between $a = 1$ and $b = 9$.

First, we define the function and sample it evenly between 0 and 10 at 200 points:

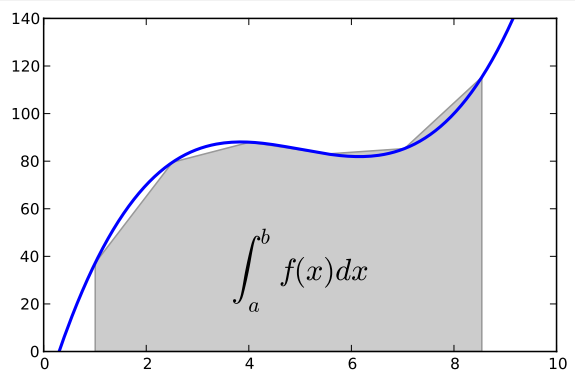
```
def f(x):
    return (x-3)*(x-5)*(x-7)+85

import numpy as np
x = np.linspace(0, 10, 200)
y = f(x)
```

We select a and b, our integration limits, and we take only a few points in that region to illustrate the error behavior of the trapezoid approximation:

```
a, b = 1, 9
xint = x[logical_and(x>=a, x<=b)][::30]
yint = y[logical_and(x>=a, x<=b)][::30]
```

Let's plot both the function and the area below it in the trapezoid approximation:

```
import matplotlib.pyplot as plt
plt.plot(x, y, lw=2)
plt.axis([0, 10, 0, 140])
plt.fill_between(xint, 0, yint, facecolor='gray', alpha=0.4)
plt.text(0.5 * (a + b), 30,r"$\int_a^b f(x)dx$", horizontalalignment='
    center', fontsize=20);
```



Compute the integral both at high accuracy and with the trapezoid approximation

```
from scipy.integrate import quad, trapz
integral, error = quad(f, 1, 9)
trap_integral = trapz(yint, xint)
print "The integral is: %g +/- %.1e" % (integral, error)
print "The trapezoid approximation with", len(xint), "points is:",
    trap_integral
print "The absolute error is:", abs(integral - trap_integral)
```

```
The integral is: 680 +/- 7.5e-12
The trapezoid approximation with 6 points is: 621.286411141
The absolute error is: 58.7135888589
```

This simple example showed us how, combining the numpy, scipy and matplotlib libraries we can provide an illustration of a standard method in elementary calculus with just a few lines of code. We will now discuss with more detail the basic usage of these tools.

3 NumPy arrays: the right data structure for scientific computing

3.1 Basics of Numpy arrays

We now turn our attention to the Numpy library, which forms the base layer for the entire 'scipy ecosystem'. Once you have installed numpy, you can import it as

```
import numpy
```

though in this book we will use the common shorthand

```
import numpy as np
```

As mentioned above, the main object provided by numpy is a powerful array. We'll start by exploring how the numpy array differs from Python lists. We start by creating a simple list and an array with the same contents of the list:

```
lst = [10, 20, 30, 40]
arr = np.array([10, 20, 30, 40])
```

Elements of a one-dimensional array are accessed with the same syntax as a list:

```
lst[0]
```

```
10
```

```
arr[0]
```

```
10
```

```
arr[-1]
```

```
40
```

```
arr[2:]
```

```
array([30, 40])
```

The first difference to note between lists and arrays is that arrays are *homogeneous*; i.e. all elements of an array must be of the same type. In contrast, lists can contain elements of arbitrary type. For example, we can change the last element in our list above to be a string:

```
lst[-1] = 'a string inside a list'
lst
```

```
[10, 20, 30, 'a string inside a list']
```

but the same can not be done with an array, as we get an error message:

```
arr[-1] = 'a string inside an array'
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
/home/fperez/teach/book-math-labtool/<ipython-input-13-29c0bfa5fa8a> in <module>()
----> 1 arr[-1] = 'a string inside an array'

ValueError: invalid literal for long() with base 10: 'a string inside an array'
```

The information about the type of an array is contained in its *dtype* attribute:

```
arr.dtype
```

```
dtype('int32')
```

Once an array has been created, its dtype is fixed and it can only store elements of the same

type. For this example where the dtype is integer, if we store a floating point number it will be automatically converted into an integer:

```
arr[-1] = 1.234
arr
```

```
array([10, 20, 30,  1])
```

Above we created an array from an existing list; now let us now see other ways in which we can create arrays, which we'll illustrate next. A common need is to have an array initialized with a constant value, and very often this value is 0 or 1 (suitable as starting value for additive and multiplicative loops respectively); `zeros` creates arrays of all zeros, with any desired dtype:

```
np.zeros(5, float)
```

```
array([ 0.,  0.,  0.,  0.,  0.])
```

```
np.zeros(3, int)
```

```
array([0, 0, 0])
```

```
np.zeros(3, complex)
```

```
array([ 0.+0.j,  0.+0.j,  0.+0.j])
```

and similarly for `ones`:

```
print '5 ones:', np.ones(5)
```

```
5 ones: [ 1.  1.  1.  1.  1.]
```

If we want an array initialized with an arbitrary value, we can create an empty array and then use the fill method to put the value we want into the array:

```
a = empty(4)
a.fill(5.5)
a
```

```
array([ 5.5,  5.5,  5.5,  5.5])
```

Numpy also offers the `arange` function, which works like the builtin `range` but returns an array instead of a list:

```
np.arange(5)
```

```
array([0, 1, 2, 3, 4])
```

and the `linspace` and `logspace` functions to create linearly and logarithmically-spaced grids respectively, with a fixed number of points and including both ends of the specified interval:

```
print "A linear grid between 0 and 1:", np.linspace(0, 1, 5)
print "A logarithmic grid between 10**1 and 10**4: ", np.logspace(1, 4, 4)
```

```
A linear grid between 0 and 1: [ 0.    0.25  0.5   0.75  1.  ]
A logarithmic grid between 10**1 and 10**4:  [    10.    100.   1000.  10000.]
```

Finally, it is often useful to create arrays with random numbers that follow a specific distribution. The `np.random` module contains a number of functions that can be used to this effect, for example this will produce an array of 5 random samples taken from a standard normal distribution (0 mean and variance 1):

```
np.random.randn(5)
```

```
array([-0.08633343, -0.67375434,  1.00589536,  0.87081651,  1.65597822])
```

whereas this will also give 5 samples, but from a normal distribution with a mean of 10 and a variance of 3:

```
norm10 = np.random.normal(10, 3, 5)
norm10
```

```
array([  8.94879575,   5.53038269,   8.24847281,  12.14944165,  11.56209294])
```

3.2 Indexing with other arrays

Above we saw how to index arrays with single numbers and slices, just like Python lists. But arrays allow for a more sophisticated kind of indexing which is very powerful: you can index an array with another array, and in particular with an array of boolean values. This is particluarly useful to extract information from an array that matches a certain condition.

Consider for example that in the array `norm10` we want to replace all values above 9 with the value 0. We can do so by first finding the *mask* that indicates where this condition is true or false:

```
mask = norm10 > 9
mask
```

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

Now that we have this mask, we can use it to either read those values or to reset them to 0:

```
print 'Values above 9:', norm10[mask]
```

```
Values above 9: [ 12.14944165  11.56209294]
```

```
print 'Resetting all values above 9 to 0...'
norm10[mask] = 0
print norm10
```

```
Resetting all values above 9 to 0...
[ 8.94879575  5.53038269  8.24847281  0.          0.        ]
```

3.3 Arrays with more than one dimension

Up until now all our examples have used one-dimensional arrays. But Numpy can create arrays of aribtrary dimensions, and all the methods illustrated in the previous section work with more than one dimension. For example, a list of lists can be used to initialize a two dimensional array:

```
lst2 = [[1, 2], [3, 4]]
arr2 = np.array([[1, 2], [3, 4]])
arr2
```

```
array([[1, 2],
       [3, 4]])
```

With two-dimensional arrays we start seeing the power of numpy: while a nested list can be indexed using repeatedly the `[ ]` operator, multidimensional arrays support a much more natural indexing syntax with a single `[ ]` and a set of indices separated by commas:

```
print lst2[0][1]
print arr2[0,1]
```

```
2
2
```

Most of the array creation functions listed above can be used with more than one dimension, for example:

```
np.zeros((2,3))
```

```
array([[ 0.,  0.,  0.],
       [ 0.,  0.,  0.]])
```

```
np.random.normal(10, 3, (2, 4))
```

```
array([[ 11.26788826,   4.29619866,  11.09346496,   9.73861307],
       [ 10.54025996,   9.5146268 ,  10.80367214,  13.62204505]])
```

In fact, the shape of an array can be changed at any time, as long as the total number of elements is unchanged. For example, if we want a 2x4 array with numbers increasing from 0, the easiest way to create it is:

```
arr = np.arange(8).reshape(2,4)
print arr
```

```
[[0 1 2 3]
 [4 5 6 7]]
```

With multidimensional arrays, you can also use slices, and you can mix and match slices and single indices in the different dimensions (using the same array as above):

```
print 'Slicing in the second row:', arr[1, 2:4]
print 'All rows, third column   :', arr[:, 2]
```

```
Slicing in the second row: [6 7]
All rows, third column   : [2 6]
```

If you only provide one index, then you will get an array with one less dimension containing that row:

```
print 'First row:  ', arr[0]
print 'Second row: ', arr[1]
```

```
First row:   [0 1 2 3]
Second row:  [4 5 6 7]
```

Now that we have seen how to create arrays with more than one dimension, it's a good idea to look at some of the most useful properties and methods that arrays have. The following provide basic information about the size, shape and data in the array:

```
print 'Data type                :', arr.dtype
print 'Total number of elements :', arr.size
print 'Number of dimensions     :', arr.ndim
print 'Shape (dimensionality)   :', arr.shape
print 'Memory used (in bytes)   :', arr.nbytes
```

```
Data type                : int32
Total number of elements : 8
Number of dimensions     : 2
Shape (dimensionality)   : (2, 4)
Memory used (in bytes)   : 32
```

Arrays also have many useful methods, some especially useful ones are:

```
print 'Minimum and maximum             :', arr.min(), arr.max()
print 'Sum and product of all elements :', arr.sum(), arr.prod()
print 'Mean and standard deviation     :', arr.mean(), arr.std()
```

```
Minimum and maximum             : 0 7
Sum and product of all elements : 28 0
Mean and standard deviation     : 3.5 2.29128784748
```

For these methods, the above operations area all computed on all the elements of the array. But for a multidimensional array, it's possible to do the computation along a single dimension, by passing the `axis` parameter; for example:

```
print 'For the following array:\n', arr
print 'The sum of elements along the rows is    :', arr.sum(axis=1)
print 'The sum of elements along the columns is :', arr.sum(axis=0)
```

```
For the following array:
[[0 1 2 3]
 [4 5 6 7]]
The sum of elements along the rows is    : [ 6 22]
The sum of elements along the columns is : [ 4  6  8 10]
```

As you can see in this example, the value of the `axis` parameter is the dimension which will be *consumed* once the operation has been carried out. This is why to sum along the rows we use `axis=0`.

This can be easily illustrated with an example that has more dimensions; we create an array with 4 dimensions and shape `(3,4,5,6)` and sum along the axis number 2 (i.e. the *third* axis, since in Python all counts are 0-based). That consumes the dimension whose length was 5, leaving us with a new array that has shape `(3,4,6)`:

```
np.zeros((3,4,5,6)).sum(2).shape
```

```
(3, 4, 6)
```

Another widely used property of arrays is the `.T` attribute, which allows you to access the transpose of the array:

```
print 'Array:\n', arr
print 'Transpose:\n', arr.T
```

```
Array:
[[0 1 2 3]
 [4 5 6 7]]
Transpose:
[[0 4]
 [1 5]
 [2 6]
 [3 7]]
```

We don't have time here to look at all the methods and properties of arrays, here's a complete list. Simply try exploring some of these IPython to learn more, or read their description in the full Numpy documentation:

```
arr.T             arr.copy          arr.getfield      arr.put           arr.squeeze
arr.all           arr.ctypes        arr.imag          arr.ravel         arr.std
arr.any           arr.cumprod       arr.item          arr.real          arr.strides
arr.argmax        arr.cumsum        arr.itemset       arr.repeat        arr.sum
arr.argmin        arr.data          arr.itemsize      arr.reshape       arr.swapaxes
arr.argsort       arr.diagonal      arr.max           arr.resize        arr.take
arr.astype        arr.dot           arr.mean          arr.round         arr.tofile
arr.base          arr.dtype         arr.min           arr.searchsorted  arr.tolist
arr.byteswap      arr.dump          arr.nbytes        arr.setasflat     arr.tostring
```

```
arr.choose        arr.dumps         arr.ndim          arr.setfield      arr.trace
arr.clip          arr.fill          arr.newbyteorder  arr.setflags      arr.transpose
arr.compress      arr.flags         arr.nonzero       arr.shape         arr.var
arr.conj          arr.flat          arr.prod          arr.size          arr.view
arr.conjugate     arr.flatten       arr.ptp           arr.sort
```

3.4 Operating with arrays

Arrays support all regular arithmetic operators, and the numpy library also contains a complete collection of basic mathematical functions that operate on arrays. It is important to remember that in general, all operations with arrays are applied *element-wise*, i.e., are applied to all the elements of the array at the same time. Consider for example:

```
arr1 = np.arange(4)
arr2 = np.arange(10, 14)
print arr1, '+', arr2, '=', arr1+arr2
```

```
[0 1 2 3] + [10 11 12 13] = [10 12 14 16]
```

Importantly, you must remember that even the multiplication operator is by default applied element-wise, it is *not* the matrix multiplication from linear algebra (as is the case in Matlab, for example):

```
print arr1, '*', arr2, '=', arr1*arr2
```

```
[0 1 2 3] * [10 11 12 13] = [ 0 11 24 39]
```

While this means that in principle arrays must always match in their dimensionality in order for an operation to be valid, numpy will *broadcast* dimensions when possible. For example, suppose that you want to add the number 1.5 to `arr1`; the following would be a valid way to do it:

```
arr1 + 1.5*np.ones(4)
```

```
array([ 1.5,  2.5,  3.5,  4.5])
```

But thanks to numpy's broadcasting rules, the following is equally valid:

```
arr1 + 1.5
```

```
array([ 1.5,  2.5,  3.5,  4.5])
```

In this case, numpy looked at both operands and saw that the first (`arr1`) was a one-dimensional array of length 4 and the second was a scalar, considered a zero-dimensional object. The broadcasting rules allow numpy to:

- *create* new dimensions of length 1 (since this doesn't change the size of the array)
- 'stretch' a dimension of length 1 that needs to be matched to a dimension of a different size.

So in the above example, the scalar 1.5 is effectively:

- first 'promoted' to a 1-dimensional array of length 1
- then, this array is 'stretched' to length 4 to match the dimension of `arr1`.

After these two operations are complete, the addition can proceed as now both operands are one-dimensional arrays of length 4.

This broadcasting behavior is in practice enormously powerful, especially because when numpy broadcasts to create new dimensions or to 'stretch' existing ones, it doesn't actually replicate the data. In the example above the operation is carried *as if* the 1.5 was a 1-d array with 1.5 in all of its entries, but no actual array was ever created. This can save lots of memory in cases when the arrays in question are large and can have significant performance implications.

The general rule is: when operating on two arrays, NumPy compares their shapes element-wise. It starts with the trailing dimensions, and works its way forward, creating dimensions of length 1 as needed. Two dimensions are considered compatible when

- they are equal to begin with, or
- one of them is 1; in this case numpy will do the 'stretching' to make them equal.

If these conditions are not met, a `ValueError: frames are not aligned` exception is thrown, indicating that the arrays have incompatible shapes. The size of the resulting array is the maximum size along each dimension of the input arrays.

This shows how the broadcasting rules work in several dimensions:

```
b = np.array([2, 3, 4, 5])
print arr, '\n\n+', b , '\n----------------\n', arr + b
```

```
[[0 1 2 3]
 [4 5 6 7]]

+ [2 3 4 5]
----------------
[[ 2  4  6  8]
 [ 6  8 10 12]]
```

Now, how could you use broadcasting to say add `[4, 6]` along the rows to `arr` above? Simply performing the direct addition will produce the error we previously mentioned:

```
c = np.array([4, 6])
arr + c
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
/home/fperez/teach/book-math-labtool/<ipython-input-45-62aa20ac1980> in <module>()
      1 c = np.array([4, 6])
----> 2 arr + c

ValueError: operands could not be broadcast together with shapes (2,4) (2)
```

According to the rules above, the array `c` would need to have a *trailing* dimension of 1 for the broadcasting to work. It turns out that numpy allows you to 'inject' new dimensions anywhere into an array on the fly, by indexing it with the special object `np.newaxis`:

```
(c[:, np.newaxis]).shape
```

```
(2, 1)
```

This is exactly what we need, and indeed it works:

```
arr + c[:, np.newaxis]
```

```
array([[ 4,  5,  6,  7],
       [10, 11, 12, 13]])
```

For the full broadcasting rules, please see the official Numpy docs, which describe them in detail and with more complex examples.

As we mentioned before, Numpy ships with a full complement of mathematical functions that work on entire arrays, including logarithms, exponentials, trigonometric and hyperbolic trigonometric functions, etc. Furthermore, scipy ships a rich special function library in the `scipy.special` module that includes Bessel, Airy, Fresnel, Laguerre and other classical special functions. For example, sampling the sine function at 100 points between 0 and 2π is as simple as:

```
x = np.linspace(0, 2*np.pi, 100)
y = np.sin(x)
```

3.5 Linear algebra in numpy

Numpy ships with a basic linear algebra library, and all arrays have a `dot` method whose behavior is that of the scalar dot product when its arguments are vectors (one-dimensional arrays) and the traditional matrix multiplication when one or both of its arguments are two-dimensional arrays:

```
v1 = np.array([2, 3, 4])
v2 = np.array([1, 0, 1])
print v1, '.', v2, '=', v1.dot(v2)
```

```
[2 3 4] . [1 0 1] = 6
```

Here is a regular matrix-vector multiplication, note that the array `v1` should be viewed as a *column* vector in traditional linear algebra notation; numpy makes no distinction between row and column vectors and simply verifies that the dimensions match the required rules of matrix multiplication, in this case we have a 2×3 matrix multiplied by a 3-vector, which produces a 2-vector:

```
A = np.arange(6).reshape(2, 3)
print A, 'x', v1, '=', A.dot(v1)
```

```
[[0 1 2]
 [3 4 5]] x [2 3 4] = [11 38]
```

For matrix-matrix multiplication, the same dimension-matching rules must be satisfied, e.g. consider the difference between $A \times A^T$:

```
print A.dot(A.T)
```

```
[[ 5 14]
 [14 50]]
```

and $A^T \times A$:

```
print A.T.dot(A)
```

```
[[ 9 12 15]
 [12 17 22]
 [15 22 29]]
```

Furthermore, the `numpy.linalg` module includes additional functionality such as determinants, matrix norms, Cholesky, eigenvalue and singular value decompositions, etc. For even more linear algebra tools, `scipy.linalg` contains the majority of the tools in the classic LAPACK libraries as well as functions to operate on sparse matrices. We refer the reader to the Numpy and Scipy documentations for additional details on these.

3.6 Reading and writing arrays to disk

Numpy lets you read and write arrays into files in a number of ways. In order to use these tools well, it is critical to understand the difference between a *text* and a *binary* file containing numerical data. In a text file, the number π could be written as "3.141592653589793", for example: a string of digits that a human can read, with in this case 15 decimal digits. In contrast, that same number written to a binary file would be encoded as 8 characters (bytes) that are not readable by a human but which contain the exact same data that the variable `pi` had in the computer's memory.

The tradeoffs between the two modes are thus:

- Text mode: occupies more space, precision can be lost (if not all digits are written to disk), but is readable and editable by hand with a text editor. Can *only* be used for one- and two-dimensional arrays.
- Binary mode: compact and exact representation of the data in memory, can't be read or edited by hand. Arrays of any size and dimensionality can be saved and read without loss of information.

First, let's see how to read and write arrays in text mode. The `np.savetxt` function saves an array to a text file, with options to control the precision, separators and even adding a header:

```
arr = np.arange(10).reshape(2, 5)
np.savetxt('test.out', arr, fmt='%.2e', header="My dataset")
!cat test.out
```

```
# My dataset
0.00e+00 1.00e+00 2.00e+00 3.00e+00 4.00e+00
5.00e+00 6.00e+00 7.00e+00 8.00e+00 9.00e+00
```

And this same type of file can then be read with the matching `np.loadtxt` function:

```
arr2 = np.loadtxt('test.out')
print arr2
```

```
[[ 0.  1.  2.  3.  4.]
 [ 5.  6.  7.  8.  9.]]
```

For binary data, Numpy provides the `np.save` and `np.savez` routines. The first saves a single array to a file with `.npy` extension, while the latter can be used to save a *group* of arrays into a single file with `.npz` extension. The files created with these routines can then be read with the `np.load` function.

Let us first see how to use the simpler `np.save` function to save a single array:

```
np.save('test.npy', arr2)
# Now we read this back
arr2n = np.load('test.npy')
# Let's see if any element is non-zero in the difference.
# A value of True would be a problem.
print 'Any differences?', np.any(arr2-arr2n)
```

```
Any differences? False
```

Now let us see how the `np.savez` function works. You give it a filename and either a sequence of arrays or a set of keywords. In the first mode, the function will auotmatically name the saved arrays in the archive as `arr_0`, `arr_1`, etc:

```
np.savez('test.npz', arr, arr2)
arrays = np.load('test.npz')
arrays.files
```

```
['arr_1', 'arr_0']
```

Alternatively, we can explicitly choose how to name the arrays we save:

```
np.savez('test.npz', array1=arr, array2=arr2)
arrays = np.load('test.npz')
arrays.files
```

```
['array2', 'array1']
```

The object returned by `np.load` from an `.npz` file works like a dictionary, though you can also access its constituent files by attribute using its special `.f` field; this is best illustrated with an example with the `arrays` object from above:

```
print 'First row of first array:', arrays['array1'][0]
# This is an equivalent way to get the same field
print 'First row of first array:', arrays.f.array1[0]
```

```
First row of first array: [0 1 2 3 4]
First row of first array: [0 1 2 3 4]
```

This `.npz` format is a very convenient way to package compactly and without loss of information, into a single file, a group of related arrays that pertain to a specific problem. At some point, however, the complexity of your dataset may be such that the optimal approach is to use one of the standard formats in scientific data processing that have been designed to handle complex datasets, such as NetCDF or HDF5.

Fortunately, there are tools for manipulating these formats in Python, and for storing data in other ways such as databases. A complete discussion of the possibilities is beyond the scope of this discussion, but of particular interest for scientific users we at least mention the following:

- The `scipy.io` module contains routines to read and write Matlab files in `.mat` format and files in the NetCDF format that is widely used in certain scientific disciplines.
- For manipulating files in the HDF5 format, there are two excellent options in Python: The PyTables project offers a high-level, object oriented approach to manipulating HDF5 datasets, while the h5py project offers a more direct mapping to the standard HDF5 library interface. Both are excellent tools; if you need to work with HDF5 datasets you should read some of their documentation and examples and decide which approach is a better match for your needs.

4 High quality data visualization with Matplotlib

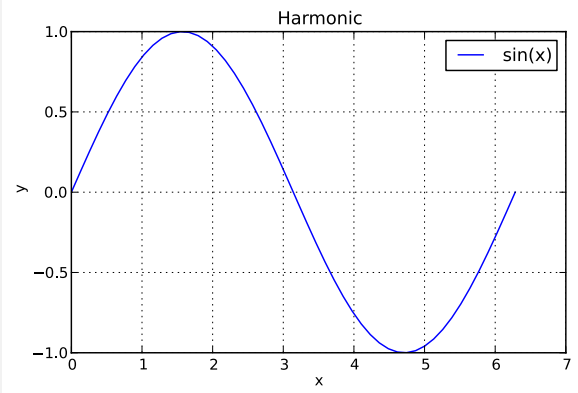
The matplotlib library is a powerful tool capable of producing complex publication-quality figures with fine layout control in two and three dimensions; here we will only provide a minimal self-contained introduction to its usage that covers the functionality needed for the rest of the book. We encourage the reader to read the tutorials included with the matplotlib documentation as well as to browse its extensive gallery of examples that include source code.

Just as we typically use the shorthand `np` for Numpy, we will use `plt` for the `matplotlib.pyplot` module where the easy-to-use plotting functions reside (the library contains a rich object-oriented architecture that we don't have the space to discuss here):

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

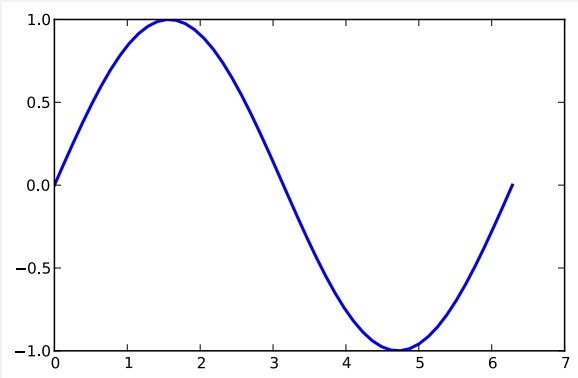
The most frequently used function is simply called `plot`, here is how you can make a simple plot of $\sin(x)$ for $x \in [0, 2\pi]$ with labels and a grid (we use the semicolon in the last line to suppress the display of some information that is unnecessary right now):

```
x = np.linspace(0, 2*np.pi)
y = np.sin(x)
plt.plot(x,y, label='sin(x)')
plt.legend()
plt.grid()
plt.title('Harmonic')
plt.xlabel('x')
plt.ylabel('y');
```

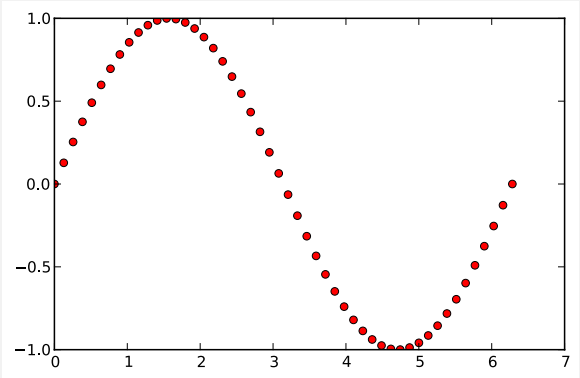


You can control the style, color and other properties of the markers, for example:

```
plt.plot(x, y, linewidth=2);
```

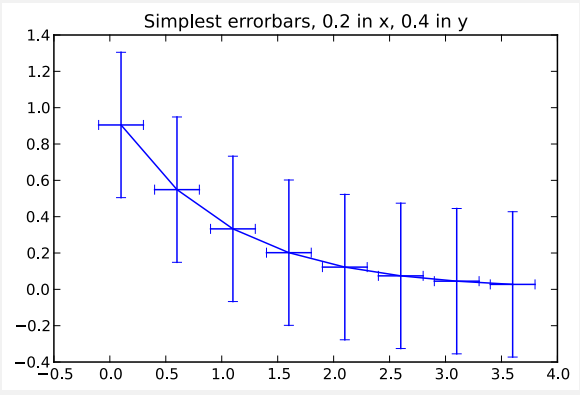



```
plt.plot(x, y, 'o', markersize=5, color='r');
```



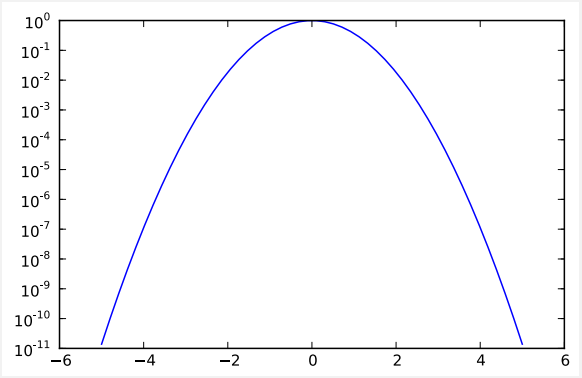
We will now see how to create a few other common plot types, such as a simple error plot:

```
# example data
x = np.arange(0.1, 4, 0.5)
y = np.exp(-x)

# example variable error bar values
yerr = 0.1 + 0.2*np.sqrt(x)
xerr = 0.1 + yerr

# First illustrate basic pyplot interface, using defaults where possible.
plt.figure()
plt.errorbar(x, y, xerr=0.2, yerr=0.4)
plt.title("Simplest errorbars, 0.2 in x, 0.4 in y");
```



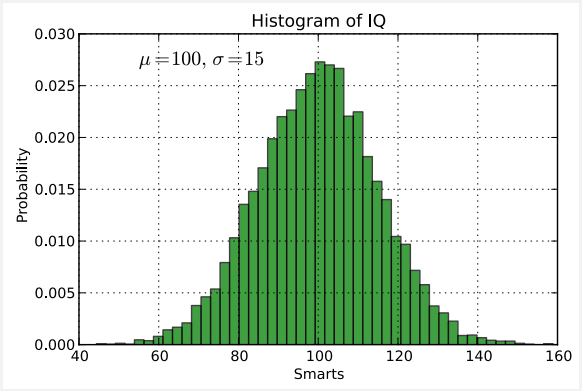
A simple log plot

```
x = np.linspace(-5, 5)
y = np.exp(-x**2)
plt.semilogy(x, y);
```



A histogram annotated with text inside the plot, using the `text` function:

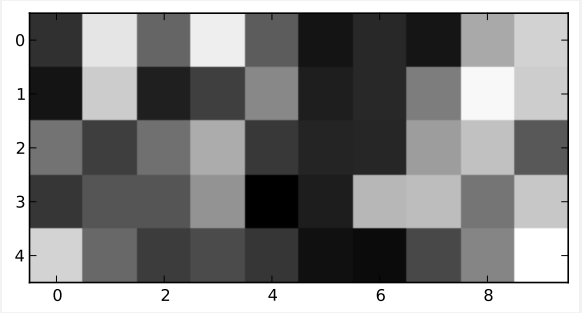
```
mu, sigma = 100, 15
x = mu + sigma * np.random.randn(10000)

# the histogram of the data
n, bins, patches = plt.hist(x, 50, normed=1, facecolor='g', alpha=0.75)

plt.xlabel('Smarts')
plt.ylabel('Probability')
plt.title('Histogram of IQ')
# This will put a text fragment at the position given:
plt.text(55, .027, r'$\mu=100,\ \sigma=15$', fontsize=14)
plt.axis([40, 160, 0, 0.03])
plt.grid(True)
```



4.1 Image display

The `imshow` command can display single or multi-channel images. A simple array of random numbers, plotted in grayscale:

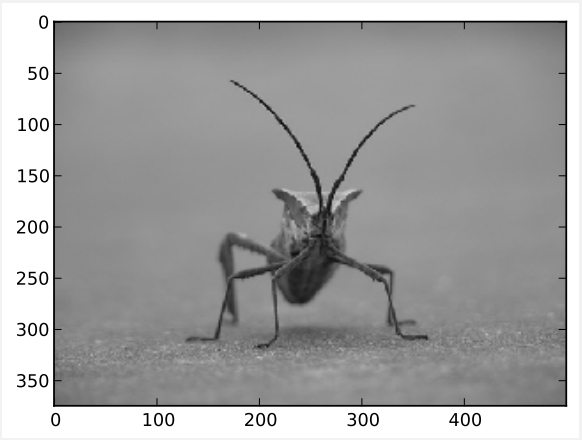
```
from matplotlib import cm
plt.imshow(np.random.rand(5, 10), cmap=cm.gray, interpolation='nearest');
```



A real photograph is a multichannel image, `imshow` interprets it correctly:

```
img = plt.imread('stinkbug.png')
print 'Dimensions of the array img:', img.shape
plt.imshow(img);
```

```
Dimensions of the array img: (375, 500, 3)
```



4.2 Simple 3d plotting with matplotlib

Note that you must execute at least once in your session:

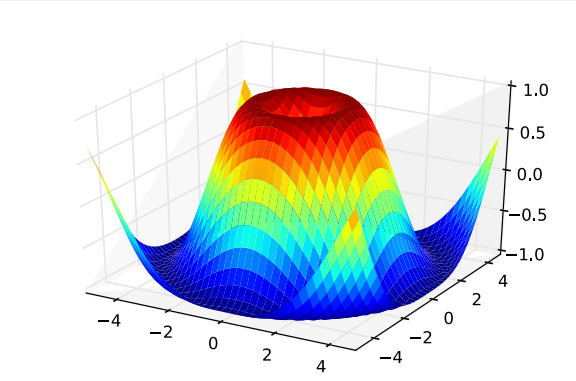
```
from mpl_toolkits.mplot3d import Axes3D
```

One this has been done, you can create 3d axes with the `projection='3d'` keyword to `add_subplot`:

```
fig = plt.figure()
fig.add_subplot(<other arguments here>, projection='3d')
```

A simple surface plot:

```
from mpl_toolkits.mplot3d.axes3d import Axes3D
from matplotlib import cm

fig = plt.figure()
ax = fig.add_subplot(1, 1, 1, projection='3d')
X = np.arange(-5, 5, 0.25)
Y = np.arange(-5, 5, 0.25)
X, Y = np.meshgrid(X, Y)
R = np.sqrt(X**2 + Y**2)
Z = np.sin(R)
surf = ax.plot_surface(X, Y, Z, rstride=1, cstride=1, cmap=cm.jet,
        linewidth=0, antialiased=False)
ax.set_zlim3d(-1.01, 1.01);
```



5 IPython: a powerful interactive environment

A key component of the everyday workflow of most scientific computing environments is a good interactive environment, that is, a system in which you can execute small amounts of code and view the results immediately, combining both printing out data and opening graphical visualizations. All modern systems for scientific computing, commercial and open source, include such functionality.

Out of the box, Python also offers a simple interactive shell with very limited capabilities. But just like the scientific community built Numpy to provide arrays suited for scientific work (since Pytyhon's lists aren't optimal for this task), it has also developed an interactive environment much more sophisticated than the built-in one. The IPython project offers a set of tools to make productive use of the Python language, all the while working interactively and with immedate feedback on your results. The basic tools that IPython provides are:

1. A powerful terminal shell, with many features designed to increase the fluidity and productivity of everyday scientific workflows, including:

   - rich introspection of all objects and variables including easy access to the source code of any function
   - powerful and extensible tab completion of variables and filenames,
   - tight integration with matplotlib, supporting interactive figures that don't block the terminal,

- direct access to the filesystem and underlying operating system,
- an extensible system for shell-like commands called 'magics' that reduce the work needed to perform many common tasks,
- tools for easily running, timing, profiling and debugging your codes,
- syntax highlighted error messages with much more detail than the default Python ones,
- logging and access to all previous history of inputs, including across sessions

2. A Qt console that provides the look and feel of a terminal, but adds support for inline figures, graphical calltips, a persistent session that can survive crashes (even segfaults) of the kernel process, and more.

3. A web-based notebook that can execute code and also contain rich text and figures, mathematical equations and arbitrary HTML. This notebook presents a document-like view with cells where code is executed but that can be edited in-place, reordered, mixed with explanatory text and figures, etc.

4. A high-performance, low-latency system for parallel computing that supports the control of a cluster of IPython engines communicating over a network, with optimizations that minimize unnecessary copying of large objects (especially numpy arrays).

We will now discuss the highlights of the tools 1–3 above so that you can make them an effective part of your workflow. The topic of parallel computing is beyond the scope of this document, but we encourage you to read the extensive documentation and tutorials on this available on the IPython website.

5.1 The IPython terminal

You can start IPython at the terminal simply by typing:

```
$ ipython
```

which will provide you some basic information about how to get started and will then open a prompt labeled `In [1]:` for you to start typing. Here we type 2^{64} and Python computes the result for us in exact arithmetic, returning it as `Out[1]`:

```
$ ipython
Python 2.7.2+ (default, Oct  4 2011, 20:03:08)
Type "copyright", "credits" or "license" for more information.

IPython 0.13.dev -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object', use 'object??' for extra details.

In [1]: 2**64
Out[1]: 18446744073709551616L
```

The first thing you should know about IPython is that all your inputs and outputs are saved. There are two variables named `In` and `Out` which are filled as you work with your results. Furthermore,

all outputs are also saved to auto-created variables of the form `_NN` where `NN` is the prompt number, and inputs to `_iNN`. This allows you to recover quickly the result of a prior computation by referring to its number even if you forgot to store it as a variable. For example, later on in the above session you can do:

```
In [6]: print _1
18446744073709551616
```

We strongly recommend that you take a few minutes to read at least the basic introduction provided by the `?` command, and keep in mind that the `%quickref` command at all times can be used as a quick reference "cheat sheet" of the most frequently used features of IPython.

At the IPython prompt, any valid Python code that you type will be executed similarly to the default Python shell (though often with more informative feedback). But since IPython is a *superset* of the default Python shell; let's have a brief look at some of its additional functionality.

Object introspection

A simple `?` command provides a general introduction to IPython, but as indicated in the banner above, you can use the `?` syntax to ask for details about any object. For example, if we type `_1?`, IPython will print the following details about this variable:

```
In [14]: _1?
Type:       long
Base Class: <type 'long'>
String Form:18446744073709551616
Namespace:  Interactive
Docstring:
long(x[, base]) -> integer

Convert a string or number to a long integer, if possible.  A floating

[etc... snipped for brevity]
```

If you add a second `?` and for any oobject `x` type `x??`, IPython will try to provide an even more detailed analsysi of the object, including its syntax-highlighted source code when it can be found. It's possible that `x??` returns the same information as `x?`, but in many cases `x??` will indeed provide additional details.

Finally, the `?` syntax is also useful to search *namespaces* with wildcards. Suppose you are wondering if there is any function in Numpy that may do text-related things; with `np.*txt*?`, IPython will print all the names in the `np` namespace (our Numpy shorthand) that have 'txt' anywhere in their name:

```
In [17]: np.*txt*?
np.genfromtxt
np.loadtxt
np.mafromtxt
np.ndfromtxt
np.recfromtxt
np.savetxt
```

Tab completion

IPython makes the tab key work extra hard for you as a way to rapidly inspect objects and libraries. Whenever you have typed something at the prompt, by hitting the `<tab>` key IPython

will try to complete the rest of the line. For this, IPython will analyze the text you had so far and try to search for Python data or files that may match the context you have already provided.

For example, if you type `np.load` and hit the key, you'll see:

```
In [21]: np.load<TAB HERE>
np.load     np.loads    np.loadtxt
```

so you can quickly find all the load-related functionality in numpy. Tab completion works even for function arguments, for example consider this function definition:

```
In [20]: def f(x, frobinate=False):
   ....:     if frobinate:
   ....:         return x**2
   ....:
```

If you now use the `<tab>` key after having typed 'fro' you'll get all valid Python completions, but those marked with `=` at the end are known to be keywords of your function:

```
In [21]: f(2, fro<TAB HERE>
frobinate=    frombuffer    fromfunction  frompyfunc    fromstring
from          fromfile      fromiter      fromregex     frozenset
```

at this point you can add the `b` letter and hit `<tab>` once more, and IPython will finish the line for you:

```
In [21]: f(2, frobinate=
```

As a beginner, simply get into the habit of using `<tab>` after most objects; it should quickly become second nature as you will see how helps keep a fluid workflow and discover useful information. Later on you can also customize this behavior by writing your own completion code, if you so desire.

Matplotlib integration

One of the most useful features of IPython for scientists is its tight integration with matplotlib: at the terminal IPython lets you open matplotlib figures without blocking your typing (which is what happens if you try to do the same thing at the default Python shell), and in the Qt console and notebook you can even view your figures embedded in your workspace next to the code that created them.

The matplotlib support can be either activated when you start IPython by passing the `--pylab` flag, or at any point later in your session by using the `%pylab` command. If you start IPython with `--pylab`, you'll see something like this (note the extra message about pylab):

```
$ ipython --pylab
Python 2.7.2+ (default, Oct  4 2011, 20:03:08)
Type "copyright", "credits" or "license" for more information.

IPython 0.13.dev -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object', use 'object??' for extra details.

Welcome to pylab, a matplotlib-based Python environment [backend: Qt4Agg].
For more information, type 'help(pylab)'.

In [1]:
```

Furthermore, IPython will import `numpy` with the `np` shorthand, `matplotlib.pyplot` as `plt`, and it will also load all of the numpy and pyplot top-level names so that you can directly type something like:

```
In [1]: x = linspace(0, 2*pi, 200)

In [2]: plot(x, sin(x))
Out[2]: [<matplotlib.lines.Line2D at 0x9e7c16c>]
```

instead of having to prefix each call with its full signature (as we have been doing in the examples thus far):

```
In [3]: x = np.linspace(0, 2*np.pi, 200)

In [4]: plt.plot(x, np.sin(x))
Out[4]: [<matplotlib.lines.Line2D at 0x9e900ac>]
```

This shorthand notation can be a huge time-saver when working interactively (it's a few characters but you are likely to type them hundreds of times in a session). But we should note that as you develop persistent scripts and notebooks meant for reuse, it's best to get in the habit of using the longer notation (known as *fully qualified names* as it's clearer where things come from and it makes for more robust, readable and maintainable code in the long run).

Access to the operating system and files

In IPython, you can type `ls` to see your files or `cd` to change directories, just like you would at a regular system prompt:

```
In [2]: cd tests
/home/fperez/ipython/nbconvert/tests

In [3]: ls test.*
test.aux  test.html  test.ipynb  test.log  test.out  test.pdf  test.rst  test.tex
```

Furthermore, if you use the `!` at the beginning of a line, any commands you pass afterwards go directly to the operating system:

```
In [4]: !echo "Hello IPython"
Hello IPython
```

IPython offers a useful twist in this feature: it will substitute in the command the value of any *Python* variable you may have if you prepend it with a `$` sign:

```
In [5]: message = 'IPython interpolates from Python to the shell'

In [6]: !echo $message
IPython interpolates from Python to the shell
```

This feature can be extremely useful, as it lets you combine the power and clarity of Python for complex logic with the immediacy and familiarity of many shell commands. Additionally, if you start the line with *two* `$$` signs, the output of the command will be automatically captured as a list of lines, e.g.:

```
In [10]: !!ls test.*
Out[10]:
['test.aux',
 'test.html',
```

```
 'test.ipynb',
 'test.log',
 'test.out',
 'test.pdf',
 'test.rst',
 'test.tex']
```

As explained above, you can now use this as the variable `_10`. If you directly want to capture the output of a system command to a Python variable, you can use the syntax `=!`:

```
In [11]: testfiles =! ls test.*

In [12]: print testfiles
['test.aux', 'test.html', 'test.ipynb', 'test.log', 'test.out', 'test.pdf', 'test.rst', 'test.tex']
```

Finally, the special `%alias` command lets you define names that are shorthands for system commands, so that you can type them without having to prefix them via `!` explicitly (for example, `ls` is an alias that has been predefined for you at startup).

Magic commands

IPython has a system for special commands, called 'magics', that let you control IPython itself and perform many common tasks with a more shell-like syntax: it uses spaces for delimiting arguments, flags can be set with dashes and all arguments are treated as strings, so no additional quoting is required. This kind of syntax is invalid in the Python language but very convenient for interactive typing (less parentheses, commans and quoting everywhere); IPython distinguishes the two by detecting lines that start with the `%` character.

You can learn more about the magic system by simply typing `%magic` at the prompt, which will give you a short description plus the documentation on *all* available magics. If you want to see only a listing of existing magics, you can use `%lsmagic`:

```
In [4]: lsmagic
Available magic functions:
%alias  %autocall  %autoindent  %automagic  %bookmark  %c  %cd  %colors  %config  %cpaste
%debug  %dhist  %dirs  %doctest_mode  %ds  %ed  %edit  %env  %gui  %hist  %history
%install_default_config  %install_ext  %install_profiles  %load_ext  %loadpy  %logoff  %logon
%logstart  %logstate  %logstop  %lsmagic  %macro  %magic  %notebook  %page  %paste  %pastebin
%pd  %pdb  %pdef  %pdoc  %pfile  %pinfo  %pinfo2  %pop  %popd  %pprint  %precision  %profile
%prun  %psearch  %psource  %pushd  %pwd  %pycat  %pylab  %quickref  %recall  %rehashx
%reload_ext  %rep  %rerun  %reset  %reset_selective  %run  %save  %sc  %stop  %store  %sx  %tb
%time  %timeit  %unalias  %unload_ext  %who  %who_ls  %whos  %xdel  %xmode

Automagic is ON, % prefix NOT needed for magic functions.
```

Note how the example above omitted the eplicit `%` marker and simply uses `lsmagic`. As long as the 'automagic' feature is on (which it is by default), you can omit the `%` marker as long as there is no ambiguity with a Python variable of the same name.

Running your code

While it's easy to type a few lines of code in IPython, for any long-lived work you should keep your codes in Python scripts (or in IPython notebooks, see below). Consider that you have a script, in this case trivially simple for the sake of brevity, named `simple.py`:

```
In [12]: !cat simple.py
import numpy as np
```

```
x = np.random.normal(size=100)

print 'First elment of x:', x[0]
```

The typical workflow with IPython is to use the `%run` magic to execute your script (you can omit the .py extension if you want). When you run it, the script will execute just as if it had been run at the system prompt with `python simple.py` (though since modules don't get re-executed on new imports by Python, all system initialization is essentially free, which can have a significant run time impact in some cases):

```
In [13]: run simple
First elment of x: -1.55872256289
```

Once it completes, all variables defined in it become available for you to use interactively:

```
In [14]: x.shape
Out[14]: (100,)
```

This allows you to plot data, try out ideas, etc, in a `%run`/interact/edit cycle that can be very productive. As you start understanding your problem better you can refine your script further, incrementally improving it based on the work you do at the IPython prompt. At any point you can use the `%hist` magic to print out your history without prompts, so that you can copy useful fragments back into the script.

By default, `%run` executes scripts in a completely empty namespace, to better mimic how they would execute at the system prompt with plain Python. But if you use the `-i` flag, the script will also see your interactively defined variables. This lets you edit in a script larger amounts of code that still behave as if you had typed them at the IPython prompt.

You can also get a summary of the time taken by your script with the `-t` flag; consider a different script `randsvd.py` that takes a bit longer to run:

```
In [21]: run -t randsvd.py

IPython CPU timings (estimated):
  User   :       0.38 s.
  System :       0.04 s.
Wall time:       0.34 s.
```

`User` is the time spent by the computer executing your code, while `System` is the time the operating system had to work on your behalf, doing things like memory allocation that are needed by your code but that you didn't explicitly program and that happen inside the kernel. The `Wall time` is the time on a 'clock on the wall' between the start and end of your program.

If `Wall > User+System`, your code is most likely waiting idle for certain periods. That could be waiting for data to arrive from a remote source or perhaps because the operating system has to swap large amounts of virtual memory. If you know that your code doesn't explicitly wait for remote data to arrive, you should investigate further to identify possible ways of improving the performance profile.

If you only want to time how long a single statement takes, you don't need to put it into a script as you can use the `%timeit` magic, which uses Python's `timeit` module to very carefully measure timig data; `timeit` can measure even short statements that execute extremely fast:

```
In [27]: %timeit a=1
```

```
10000000 loops, best of 3: 23 ns per loop
```

and for code that runs longer, it automatically adjusts so the overall measurement doesn't take too long:

```
In [28]: %timeit np.linalg.svd(x)
1 loops, best of 3: 310 ms per loop
```

The `%run` magic still has more options for debugging and profiling data; you should read its documentation for many useful details (as always, just type `%run?`).

5.2 The graphical Qt console

If you type at the system prompt (see the IPython website for installation details, as this requires some additional libraries):

```
$ ipython qtconsole
```

instead of opening in a terminal as before, IPython will start a graphical console that at first sight appears just like a terminal, but which is in fact much more capable than a text-only terminal. This is a specialized terminal designed for interactive scientific work, and it supports full multi-line editing with color highlighting and graphical calltips for functions, it can keep multiple IPython sessions open simultaneously in tabs, and when scripts run it can display the figures inline directly in the work area.

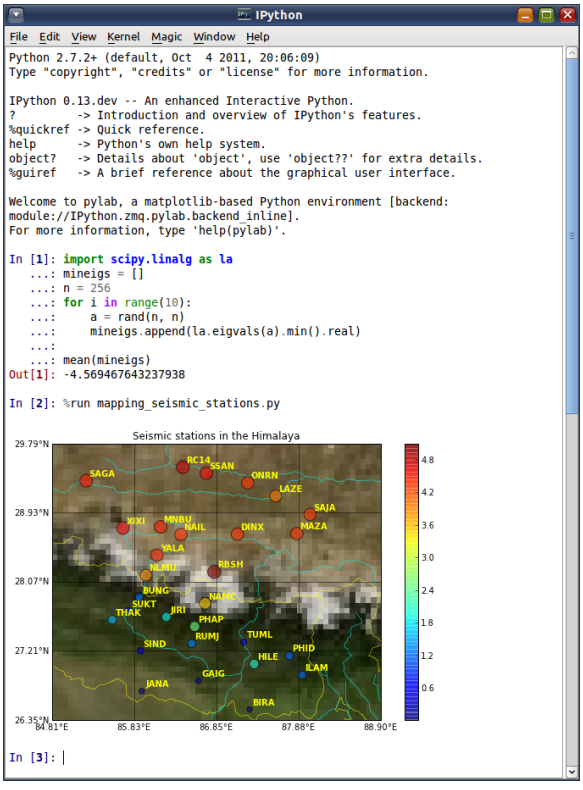


Figure 1: The IPython Qt console: a lightweight terminal for scientific exploration, with code, results and graphics in a soingle environment.

The Qt console accepts the same `--pylab` startup flags as the terminal, but you can additionally supply the value `--pylab inline`, which enables the support for inline graphics shown in the figure. This is ideal for keeping all the code and figures in the same session, given that the console can save the output of your entire session to HTML or PDF.

Since the Qt console makes it far more convenient than the terminal to edit blocks of code with multiple lines, in this environment it's worth knowing about the `%loadpy` magic function. `%loadpy` takes a path to a local file or remote URL, fetches its contents, and puts it in the work area for you to further edit and execute. It can be an extremely fast and convenient way of loading code from local disk or remote examples from sites such as the Matplotlib gallery.

Other than its enhanced capabilities for code and graphics, all of the features of IPython we've explained before remain functional in this graphical console.

5.3 The IPython Notebook

The third way to interact with IPython, in addition to the terminal and graphical Qt console, is a powerful web interface called the "IPython Notebook". If you run at the system console (you can omit the `pylab` flags if you don't need plotting support):

```
$ ipython notebook --pylab inline
```

IPython will start a process that runs a web server in your local machine and to which a web browser can connect. The Notebook is a workspace that lets you execute code in blocks called 'cells' and displays any results and figures, but which can also contain arbitrary text (including LaTeX-formatted mathematical expressions) and any rich media that a modern web browser is capable of displaying.

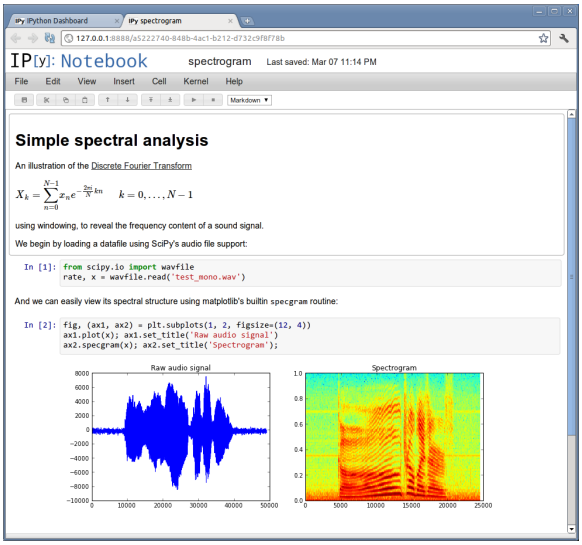


Figure 2: The IPython Notebook: text, equations, code, results, graphics and other multimedia in an open format for scientific exploration and collaboration

In fact, this document was written as a Notebook, and only exported to LaTeX for printing. Inside of each cell, all the features of IPython that we have discussed before remain functional, since ultimately this web client is communicating with the same IPython code that runs in the terminal.

But this interface is a much more rich and powerful environment for maintaining long-term "live and executable" scientific documents.

Notebook environments have existed in commercial systems like Mathematica(TM) and Maple(TM) for a long time; in the open source world the Sage project blazed this particular trail starting in 2006, and now we bring all the features that have made IPython such a widely used tool to a Notebook model.

Since the Notebook runs as a web application, it is possible to configure it for remote access, letting you run your computations on a persistent server close to your data, which you can then access remotely from any browser-equipped computer. We encourage you to read the extensive documentation provided by the IPython project for details on how to do this and many more features of the notebook.

Finally, as we said earlier, IPython also has a high-level and easy to use set of libraries for parallel computing, that let you control (interactively if desired) not just one IPython but an entire cluster of 'IPython engines'. Unfortunately a detailed discussion of these tools is beyond the scope of this text, but should you need to parallelize your analysis codes, a quick read of the tutorials and examples provided at the IPython site may prove fruitful.