

---

# PyPrimer Documentation

*Release 0.1*

**DPine**

December 02, 2011

# CONTENTS

<b>1</b>	<b>Introduction</b>	<b>2</b>
1.1	Introduction to Python and its use in science . . . . .	2
<b>2</b>	<b>Launching Python</b>	<b>3</b>
2.1	Installing Python on your computer . . . . .	3
2.2	The IPython shell . . . . .	4
2.3	Creating a new program . . . . .	5
2.4	The interactive interpreter . . . . .	5
<b>3</b>	<b>The elements of Python</b>	<b>6</b>
3.1	Interpreters, modules, and a more interesting program . . . . .	6
3.2	Variables . . . . .	7
3.3	Lists and tuples . . . . .	9
3.4	Input and output . . . . .	11
3.5	Arithmetic . . . . .	12
3.6	for loops . . . . .	13
3.7	if statements . . . . .	15
3.8	while loops . . . . .	17
3.9	Using library functions . . . . .	18
3.10	Arrays . . . . .	20
3.11	Making your own functions . . . . .	24
3.12	File input and output . . . . .	29
3.13	Putting it all together . . . . .	33
<b>4</b>	<b>Graphics and Curve Fitting</b>	<b>35</b>
4.1	Basic graphical output . . . . .	35
4.2	More advanced graphical output . . . . .	41
4.3	Curve Fitting . . . . .	43
<b>5</b>	<b>Additional Python</b>	<b>53</b>
5.1	Running Python in other shells . . . . .	53
5.2	List Comprehensions . . . . .	53
5.3	Arrays in Python . . . . .	54
5.4	Functions you may need . . . . .	56
5.5	Scope . . . . .	56
5.6	Python differences . . . . .	57
5.7	Python Modules . . . . .	58
5.8	Python language summary . . . . .	60
5.9	Taking your interest further . . . . .	62

<b>6</b>	<b>Getting Help with Python</b>	<b>64</b>
6.1	Getting help from within IPython . . . . .	64
6.2	Web Resources for Python . . . . .	65
6.3	Books about Python . . . . .	65
<b>7</b>	<b>Errors</b>	<b>67</b>
7.1	Attribute Errors, Key Errors, Index Errors . . . . .	67
7.2	Name Errors . . . . .	67
7.3	Syntax Errors . . . . .	68
7.4	Type Errors . . . . .	68
<b>8</b>	<b>Reserved Words</b>	<b>69</b>
<b>9</b>	<b>Installing Python</b>	<b>70</b>
9.1	Setting up Python on Windows XP and Windows 7 . . . . .	70
9.2	Setting up Python on MacOSX . . . . .	72

Contents:

---

# INTRODUCTION

**authors** David Pine

## 1.1 Introduction to Python and its use in science

This manual will serve as an introduction to the Python programming language and its use for scientific programming. You can use Python to analyze and plot data. You can also use it to numerically solve physics problems that are difficult or even impossible to solve analytically.

While we want to marshal Python's powers to address scientific problems, you should know that Python is a general purpose computer language that is widely used to address all kinds of computing tasks, from web applications on Google and YouTube to processing financial data on Wall Street and various scripting tasks for computer system management. Over the past decade it has been increasingly used by scientists for numerical computations, graphics, and as a "wrapper" for numerical software originally written in Fortran, C, and other languages.

Python is similar to Matlab and IDL, two other computer languages that are frequently used in scientific and engineering applications. Like Matlab and IDL, Python is an *interpreted* language, meaning you can run your code without having to go through an extra step of compiling, as required for the C and Fortran programming languages. It is also a *dynamically typed* language, meaning you don't have to declare variables and set aside memory before using them. Don't worry if you don't know exactly what these terms mean. Their primary significance for you is that you can write Python code, test, and use it quickly with a minimum of fuss.

One advantage of Python over similar languages like Matlab and IDL is that it is free. It can be downloaded from the web and is available on all the standard computer platforms, including Windows, MacOS, and Linux. This also means that you can use Python without being tethered to the internet, as required for commercial software that is tied to a remote license server.

Another advantage is Python's clean and simple syntax, including its implementation of *object oriented* programming (which we do not emphasize in this introduction).

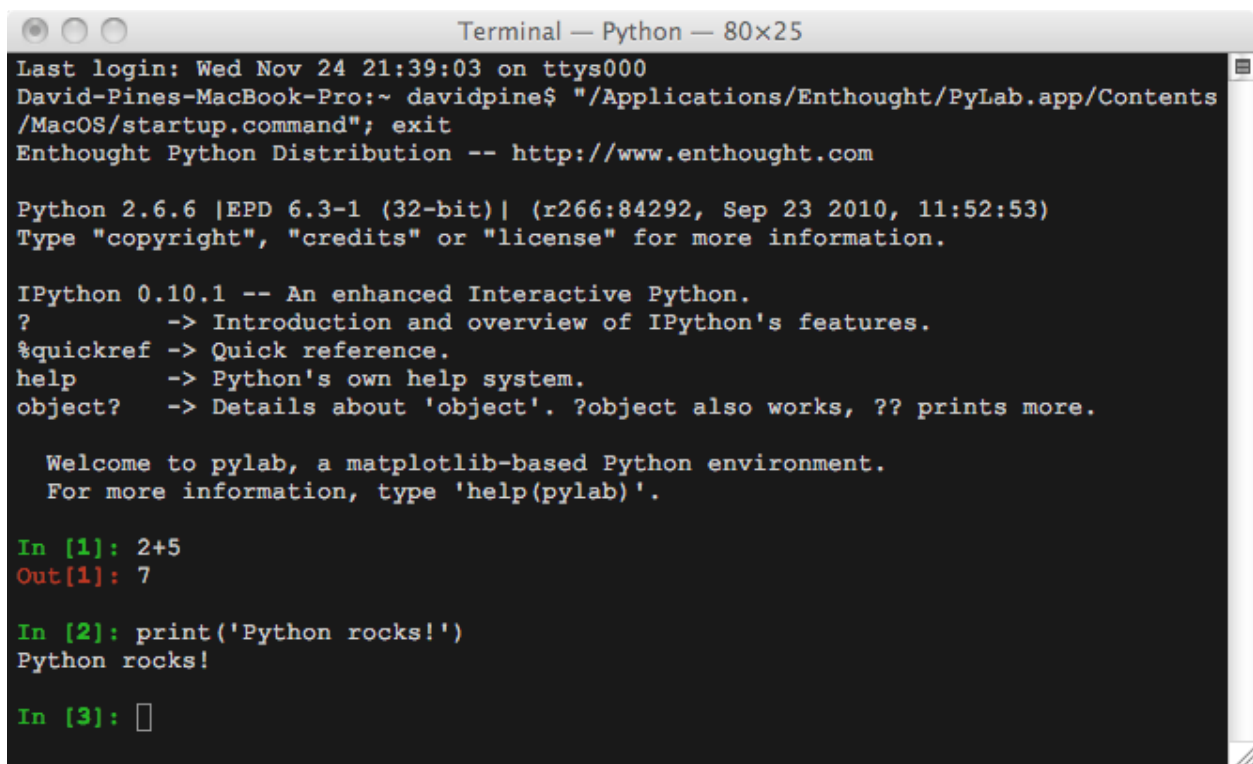
An important disadvantage is that Python programs can be considerably slower than compiled languages like C. For large scale simulations and other demanding applications, there can be a considerable speed penalty in using Python. In these cases, C, C++, or Fortran is recommended, although there are specialized tools, such as Cython (not covered in this manual), that can greatly speed up Python code. Another disadvantage is that compared to Matlab and IDL, Python is less well documented. This stems from the fact that it is public *open source* software and thus is dependent on volunteers from the community of developers and users for documentation. The documentation is freely available on the web but is scattered among a number of different sites and can be terse. This manual will acquaint you with the most commonly-used web sites. Search engines such as Google can help you find others.

# LAUNCHING PYTHON

## 2.1 Installing Python on your computer

If you haven't already installed Python on your computer, see *Installing Python*, which includes instructions for installing Python on Macs running under MacOSX and on PCs running under either Windows XP or Windows 7.

Once you have installed Python, find the IPython or PyLab icon to launch the application. Wait for a window to appear, like the one shown below. Yours may have a black letters on a white background or some other color scheme. This is the window you will generally use to work with Python.



```
Terminal — Python — 80x25
Last login: Wed Nov 24 21:39:03 on ttys000
David-Pines-MacBook-Pro:~ davidpine$ "/Applications/Enthought/PyLab.app/Contents
/MacOS/startup.command"; exit
Enthought Python Distribution -- http://www.enthought.com

Python 2.6.6 |EPD 6.3-1 (32-bit)| (r266:84292, Sep 23 2010, 11:52:53)
Type "copyright", "credits" or "license" for more information.

IPython 0.10.1 -- An enhanced Interactive Python.
?      -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help    -> Python's own help system.
object? -> Details about 'object'. ?object also works, ?? prints more.

Welcome to pylab, a matplotlib-based Python environment.
For more information, type 'help(pylab)'.

In [1]: 2+5
Out[1]: 7

In [2]: print('Python rocks!')
Python rocks!

In [3]: □
```

Figure 2.1: IPython command-line window

## 2.2 The IPython shell

For scientific computing, we usually communicate with Python through the *IPython shell*, which is an advanced command-line window (see [Figure 2.1 IPython command-line window](#)). Sometimes the IPython shell is referred to as *PyLab* (*n.b.* IPython shell = PyLab shell). We often also refer to the command-line window as the *console* or the *IPython console*.

From the IPython console you can type in commands using your computer’s keyboard. These commands tell Python to perform various tasks, like adding numbers, making plots, or performing other computational tasks. The IPython shell can also be used to perform some system tasks, such as creating and moving files around on your computer. This is a different kind of computer interface than the icon-based interface (or “graphical user interface” GUI) that you usually use to communicate with your computer. While it may seem more cumbersome for some tasks, it can be more powerful for other tasks, particularly those associated with programming. In this section, we give you a brief introduction to its use.

From the IPython shell prompt you can do many different things, but for the moment we shall concentrate simply on navigating around your computer’s directory system. IPython recognizes a number of commands from the *UNIX* or Linux operating systems. So if you are familiar with these commands, you already know what to do. Here, we introduce a few useful commands for navigating the file system of your computer. These commands work on either a Mac or a PC in the IPython console.

Like most modern computer languages, Python is *case sensitive*. That is, Python distinguishes between upper and lower case letters. Thus, two words spelled the same but having different letters capitalized are treated as different names in Python.

At a IPython prompt, type `cd ~` (*i.e.* “cd” – “space” – “tilde”, where tilde is found of the shifted key to the left of the “1” key on your keyboard). This will set your computer to its home (default) directory. Next type `pwd` (**p**rint **w**orking **d**irectory) and press RETURN. The console should return the path of the current directory of your computer. It might look like this on a Mac:

```
In [1]: pwd
Out[1]: '/Users/pine'
```

or this on a PC:

```
In [1]: pwd
Out[1]: 'C:\\My Documents and Settings\\pine'
```

Now type `ls` (**l**ist) and press RETURN. The console should list the names of the files and subdirectories in the current directory. If you just launched a newly-installed version of IPython, then among the files and directories listed you should see something like `Documents/` (or perhaps `My Documents` on some PCs). To change to this directory, use the `cd` (**c**hange **d**irectory) command: type `cd Documents/` if you are using a Mac or `cd 'My Documents'` if you are using a PC (the quotation marks are needed because `'My Documents'` contains a space). Now type `pwd`. You should see that you are now in your documents directory.

Let’s create a directory within your documents directory that you can use to store your Python programs. We will call it `PyProgs`. To create this directory, type `mkdir PyProgs` (**m**ake **d**irectory). Type `ls` to confirm that you have created `PyProgs` and then type `cd PyProgs` to switch to that directory.

IPython also incorporates a number of shortcuts that make using the shell more efficient. One of the most useful is *tab completion*; if you start typing a command, say `run test.py`, you can type `run t` and then press the TAB key, which will complete the command `run test.py` if there is no ambiguity in how to finish the command. In the present case, that would mean that there was no other Python program (*i.e.* a file ending with the `.py` suffix in the current working directory that began with the letter `t`). Try it out! It will make your life more wonderful. A related shortcut is to type a command, say `cd` and then to press the `↑` key, which will complete the `cd` command with the last instance of that command. Thus, when you launch IPython, you can use this shortcut to take you to the directory you

used when you last ran IPython. The next section explains how to create your first Python program and save it in the directory `PyProgs`.

Let's recap the commands introduced above:

- `pwd`: **print working directory** — prints the path of the current directory.
- `ls`: **list** — list the names of the files and directories located in the current directory
- `mkdir filename`: **make directory** — makes a new directory *filename*.
- `cd directoryname`: **change directory** — changes the current directory to *directoryname*. Note: for this to work, *directoryname* must be a subdirectory in the current directory. Typing `cd ~` changes to the home directory of your computer. Typing `cd ..` moves the console one directory up in the directory tree.

## 2.3 Creating a new program

Using the `cd` command, set your current directory to be the `PyProgs` directory you created. Now type `edit test.py`. It is important that you include the `.py` extension as that lets IPython (and the operating system) know that this is a Python program. If you have installed IPython and the requisite editor ( **TextWrangler** on the Mac and **Notebook++** on the PC), an editor window should be launched. On the first line in the editor, type:

```
print('Python rocks')
```

Save the program and exit. Then in the IPython shell, type:

```
run test
```

The console should then output:

```
Python rocks
```

You have written your first Python program.

## 2.4 The interactive interpreter

The IPython shell serves not only as a shell for performing system functions, such as moving about the directory structure, but also as an *interactive interpreter* for Python. This means that you can write Python commands directly at the IPython prompt, and the IPython shell will interpret them as Python commands. For example, the commands in the program you wrote and stored in the file `test.py` can be executed directly from the IPython prompt. Try it out. When you do, you should see something like:

```
In [1]: print('Python rocks')
Out[1]: Python rocks
```

You type in `print('Python rocks')`, press return, and the console writes out `Out[1]: Python rocks`. This is a very useful feature of Python that we will use again and again.



# THE ELEMENTS OF PYTHON

In this chapter we introduce the basic elements of programming in Python. We emphasize those elements that are most useful of scientific or technical programming.

## 3.1 Interpreters, modules, and a more interesting program

We saw in Chapter *Launching Python* that there are two ways of using Python: either using its *interactive interpreter* or by writing *modules*. The interpreter is useful for testing small snippets of programs. In general, all the examples you see in this manual can be typed at the interactive prompt (`In [1]:`). In fact, you should get into the habit of trying things out at the prompt: you won't harm your computer, and it is a good way of experimenting and making sure the code you are writing actually works as intended.

However, working interactively has the serious drawback that you cannot save your work. When you exit the interactive interpreter everything you have done is lost. If you want to write a longer program you need to create a *module*. This is just a text file containing a list of Python instructions. When the module is *run*, Python simply reads through it one line after another, as though it had been typed at the interactive prompt.

To create a module, just open your editor (**TextWrangler** on the Mac and **Notebook++** on the PC). You create a Python module simply by typing code into your text editor and saving the file, giving it whatever name you wish so long as you append the `.py` extension at the end, which lets the system know that this is a Python module.

Here is an example of a complete Python module. Type it into an editor window and save it to your hard disk using *Save As* from the *File* menu. Be sure to save it to the current directory on your hard disk (the directory that is returned when you type `pwd` in the console). Run the program by typing `%run` in the console:

```
1 print("Please type in a number: ")
2 a = input()
3 print("...and another: ")
4 b = input()
5 print("The sum of these numbers is: ")
6 print(a + b)
```

If you get errors, check the module you wrote for small mistakes like missing punctuation marks. Once you have run the program, it should be apparent how it works. The only thing that might not be obvious are the lines with `input()`. `input()` is a *function* that allows a user to type in a number and then store it for use in the rest of the program: in this case the inputs are stored in `a` and `b`.

If you are writing a module and you want to save your work, do so by selecting *Save* from the *File* menu then type a name for your program in the box. The name you choose should indicate what the program does and consist only of letters, numbers, and “`_`” the underscore character. The name **must end with a `.py`** so that it is recognized as a Python module, e.g. `prog.py`. Furthermore, **do NOT use spaces in filenames or directory (folder) names**.

**EXERCISE 3.1:** Change the program so it subtracts the two numbers, rather than adds them up. Be sure to test that your program works as it should.

## 3.2 Variables

### 3.2.1 Names and Assignment

In Section *Interpreters, modules, and a more interesting program* we used *variables* for the first time: `a` and `b` in the example. Variables are used to store data; in simple terms they are much like variables in algebra and, as mathematically-literate students, we hope you will find the programming equivalent fairly intuitive.

Variables have names like `a` and `b` above, or `x` or `fred` or `z1`. We recommend giving your variables a descriptive name, such as `FirstName` or `height`.<sup>1</sup> **Variable names must start with a letter and then may consist only of alphanumeric characters (i.e. letters and numbers) and the underscore character, “\_”.** There are some reserved words which you cannot use because Python uses them for other things; these are listed in Appendix *Reserved Words*.

We *assign* values to variables and then, whenever we refer to a variable later in the program, Python replaces its name with the value we assigned to it. This is best illustrated by a simple example:

```
In [15]: x=5
In [16]: print(x)
5
```

You assign a value to a variable name by putting the variable name on the left, followed by a single `=`, followed by what is to be stored. To draw an analogy, you can think of variables as named boxes. What we have done above is to label a box with an “`x`”, and then put the number 5 in that box.

There are some differences between the *syntax*<sup>2</sup> of Python and normal algebra which are important. Assignment statements read *right to left only*. `x = 5` is fine, but `5 = x` doesn’t make sense to Python, which will report a `SyntaxError`. If you like, you can think of the equals sign as an arrow pointing from the number on the right, to the variable name on the left:  $x \leftarrow 5$  and read the expression as “assign 5 to `x`”. However, we can still do many of things you might do in algebra, like:

```
In [17]: a = b = c = 0
```

Reading the above right to left we have: “assign 0 to `c`, assign `c` to `b`, assign `b` to `a`”:

```
In [18]: print(a, b, c)
(0, 0, 0)
```

There are also statements that are algebraically nonsense, that are perfectly sensible to Python (and indeed to most other programming languages). The most common example is incrementing a variable:

```
In [19]: i=2
In [20]: i=i+1
In [21]: print(i)
3
```

The second line in this example is not possible in mathematics, but makes sense in Python if you think of the equals as an arrow pointing from right to left. To describe the statement in words: on the right-hand side we have looked at what is in the box labelled `i`, added 1 to it, then stored the result back in the same box.

<sup>1</sup> Python is named after the 1970s TV series *Monty Python’s Flying Circus*, so variables in examples are often named after sketches in the series. Don’t be suprised to see `spam`, `lumberjack`, and `shrubbery` if you read up on Python on the web.

<sup>2</sup> The *syntax* of a language refers to its grammar and structure: i.e. the order and way in which things are written.

The assignment `i=i+1`, where a variable is assigned a new value based on its current value, is used so often that Python has a special shorthand for it. The code snippet below illustrates how it works:

```
In [30]: i=3
In [31]: i += 1
In [32]: print(i)
4
In [33]: i += 2
In [34]: print(i)
6
```

As you can see `i+=1` is equivalent to `i=i+1` and `i+=2` is equivalent to `i=i+2`. Similarly, you can use `j-=1` for `j=j-1`, `a/=3` for `a=a/3`, and `a*=3` for `a=a*3`. For example:

```
In [35]: a=12
In [36]: a/=3
In [37]: print(a)
4
```

### 3.2.2 Types

Your variables need not be numeric. There are several *types*. The most useful are described below:

**Integer:** Any whole number:

```
In [22]: myinteger = 0
In [23]: myinteger = 15
In [24]: myinteger = -23
In [25]: myinteger = 2378
```

**Float:** A floating point number, *i.e.* a real number.

```
In [26]: myfloat = 0.1
In [27]: myfloat = 2.0
In [28]: myfloat = 3.14159256
In [29]: myfloat = 1.6e-19
In [30]: myfloat = 3e8
```

Note that although 2 is an integer, by writing it as `2.0` we indicate that we want it stored as a floating point number, or a *float*, with the precision that entails.

The last examples use exponentials, and in math would be written  $1.6 \times 10^{-19}$  and  $3 \times 10^8$ . If the number is given in exponential form it is stored with the precision of floating point whether or not it is a whole number.

**Complex:** A complex number of the form  $4 + 6i$ . In Python the symbol `j` is used to denote  $i = \sqrt{-1}$ . In the IPython console we write:

```
In [31]: mycmplx = 4+6j
In [32]: print(mycmplx)
(4+6j)
```

**String:** A string or sequence of characters that can be printed on your screen. They must be enclosed in *either* single quotes *or* double quotes—not a mixture of the two, *e.g.* :

```
In [33]: mystring = "Here is a string"
In [34]: mystring = 'Here is another'
```

If you are not sure what type a variable is, you can use the `type()` function to inspect it:

```
In [35]: type(mystring)
<type 'str'>
```

'str' tells you it is a string. You might also get <type 'int'> (integer) and <type 'float'> (float)<sup>3</sup>

**EXERCISE 3.2:** Use the interactive interpreter to create integer, float and string variables. Once you've created them print them to see how Python stores them.

Experiment with the following code snippet to prove to yourself that Python is *case-sensitive*, i.e. whether a variable named `a` is the same as one called `A`:

```
In [36]: a = 1.2
In [37]: print(A)
```

You can use lower and upper case letters to make the variable and functions you define more readable. For example, you might write `mPend` for the mass of a pendulum or `MagDipole` for the magnetic dipole moment in some code you are writing. Some people like to use the underscore character “\_” as a separator in a variable name like `mag_dipole`. I find that inelegant (only my opinion!). Besides, as we shall see, Python uses underscore characters in certain system variables and functions so I like to avoid their use in my code.

### 3.3 Lists and tuples

There are various constructs in mathematics and physics that are lists of numbers or variables: sequences, vectors, and matrices are examples. One familiar example is the Fibonacci sequence  $\{0, 1, 1, 2, 3, 5, 8, 13, 21, \dots\}$ , formed by starting with the two-element list  $\{0, 1\}$ , and then adding additional elements generated by the sum of the two previous elements. We might write the  $i^{\text{th}}$  element of the sequence as  $f_i$ . According to our rule the next element in the sequence is given by  $f_{i+1} = f_i + f_{i-1}$ .

Python has two data structures, *lists* and *tuples*, that consist of a list of elements. Those elements can be numbers, but they can also be strings, so lists and tuples are perhaps a little more general than the mathematical lists encountered in physics. We shall also work with a data structure called an *array*, but we defer discussion of arrays until Section [Arrays](#).

Lists and tuples are part of core Python. They are data types that contain more than one element. Here are two examples of lists:

```
In [12]: a = [0, 1, 1, 2, 3, 5, 8, 13, 21]
In [13]: b = [5., 'girl', 2+0j, 'horse', 21]
```

The individual elements in list are separated by commas and can be any valid kind of Python variable. All the elements can be of the same type, as in the case of list `a` above, or they can be different types, as for the case of list `b`. We can print out the lists as follows:

```
In [14]: print(a)
[0, 1, 1, 2, 3, 5, 8, 13, 21]
In [15]: print(b)
[5.0, 'girl', (2+0j), 'horse', 21]
```

We can also access individual elements of a list:

```
In [17]: b[0]
Out[17]: 5.0

In [20]: b[1]
Out[20]: 'girl'
```

<sup>3</sup> If you've programmed in other languages before, you might be used to “declaring” variables before you use them. In Python this is not necessary. In fact there are several things Python “does for you” that may surprise you if you've programmed before. See Section [Python differences](#).

```
In [21]: b[2]
Out[21]: (2+0j)
```

Notice that the first element of the list `b` is `b[0]`, the second is `b[1]`, the third is `b[2]`, and so on. Some computer languages index lists starting with 0, like Python and C, while others index lists (or things more-or-less equivalent) starting with 1. It's important to keep in mind that Python uses the former convention: lists (and tuples and arrays) are *zero-indexed*.

Note also that the different elements of the list are accessed using *square* brackets.

The last element of this array is `b[4]`, because `b` has 5 elements. The last element can also be accessed as `b[-1]`, no matter how many elements `b` has, and the next-to-last element of the list is `b[-2]`, *etc.* Try it out:

```
In [22]: print(b[4])
21

In [23]: print(b[-1])
21

In [24]: print(b[-2])
horse
```

Individual elements of lists can be changed. For example:

```
In [25]: print(b)
Out[25]: [5.0, 'girl', (2+0j), 'horse', 21]
In [26]: b[0] = b[0]+2
In [27]: print(b)
Out[27]: [7.0, 'girl', (2+0j), 'horse', 21]
```

Here we see that 2 was added to the previous value of `b[0]`. We can even do the same thing with strings:

```
In [33]: b[1] = b[1] + 's & boys'
In [34]: print(b)
Out[34]: [10.0, 'girls & boys', (2+0j), 'horse', 21]
```

Note that in the case of strings “+” means concatenation.

You can also add lists, but the result might surprise you:

```
In [49]: print(a)
Out[49]: [0, 1, 1, 2, 3, 5, 8, 13, 21]

In [50]: print(a+a)
Out[50]: [0, 1, 1, 2, 3, 5, 8, 13, 21, 0, 1, 1, 2, 3, 5, 8, 13, 21]

In [51]: print(a+b)
Out[51]: [0, 1, 1, 2, 3, 5, 8, 13, 21, 10.0, 'girls & boys', (2+0j), 'horse', 21]
```

Thus we see that adding lists concatenates them, similar to the way the “+” operates on strings.

### 3.3.1 Slicing lists

You can access pieces of lists using the *slicing* feature of Python:

```
In [83]: print(b)
Out[83]: [5.0, 'girl', (2+0j), 'horse', 21]

In [84]: print(b[1:4])
```

```
Out[84]: ['girl', (2+0j), 'horse']

In [85]: b[3:5]
Out[85]: ['horse', 21]
```

You get a subset of a list by specifying two indices separated by a colon “:”. This is a powerful feature of lists in Python that we will use often. Here are a few other useful slicing shortcuts:

```
In [94]: print(b[2:])
Out[94]: [(2+0j), 'horse', 21]

In [95]: print(b[:3])
Out[95]: [5.0, 'girl', (2+0j)]

In [96]: print(b[:])
Out[96]: [5.0, 'girl', (2+0j), 'horse', 21]
```

Thus, if the left slice index is 0, you can leave it out; similarly, if the right slice index is the length of the list, you can leave it out also.

What does the following slice of an array give you?

```
In [97]: print(b[1:-1])
```

If you don’t know the length of a list you can get it using Python’s `len` function:

```
In [98]: len(b)
Out[98]: 5
```

### 3.3.2 Tuples

Finally, a word about tuples: tuples are lists that are *immutable*. That is, once defined, the individual elements of a tuple cannot be changed:

```
In [52]: c = (1, 1, 2, 3, 5, 8, 13)
In [53]: c[4]
Out[53]: 5

In [54]: c[4] = 7
-----
TypeError: 'tuple' object does not support item assignment
```

When we tried to change `c[4]`, the system returned an error because we are prohibited from changing an element of a tuple. Tuples offer some degree of safety when we want to define lists of immutable constants.

## 3.4 Input and output

Computer programs generally involve interaction with the user, often called input and output. Output involves printing things to the screen (and, as we shall see later, it also involves writing data to files, sending plots to a printer, *etc.*). We have already seen one way of getting input—the `input()` function in Section *Interpreters, modules, and a more interesting program*. Functions will be discussed in more detail in Section *Using library functions*, but for now we can use the `input()` function to get numbers (and only numbers) from the keyboard.

You can put a string between the parentheses of `input()` to give the user a prompt. Hence the module in Section *Interpreters, modules, and a more interesting program* could be rewritten as follows:

```
a = input("Please type in a number: ")
b = input("...and another: ")
print("The sum of these numbers is:", a+b)
```

Note that in this mode of operation the `input()` function is actually doing output as well as input!

The `print` function can print several things, which we separate with a comma, as above. If the `print` function is asked to print more than one thing, separated by commas, it separates them with a space. You can also *concatenate* (join) two strings using the `+` operator (note that no spaces are inserted between concatenated strings):

```
In [36]: x = "Spanish Inquisition"
In [37]: print("Nobody expects the" + x)
(Nobody expects theSpanish Inquisition)
```

`input()` can read in numbers (integers and floats) only. If you want to read in a string (a word or sentence for instance) from the keyboard, then you should use the `raw_input()` function; for example:

```
In [38]: name = raw_input("Please tell me your name: ")
```

**EXERCISE 3.4:** Copy the example in Section *Interpreters, modules, and a more interesting program* into an empty module. Modify it so that it reads in *three* numbers and adds them up. Further modify the program to ask the user for their name before inputting the three numbers. Then, instead of just outputting the sum, personalize the output message; e.g. by first saying: “*name* here are your results” where *name* is their name. Think carefully about when to use `input()` and `raw_input()`.

## 3.5 Arithmetic

Python performs arithmetic in the usual way, and the console acts much like a calculator. Basic calculations are expressed using Python’s (mostly obvious) *arithmetic operators*. If there is one subtlety in Python, it is that *decimal points matter*! Numbers with decimal points are floats while numbers without decimal points are integers. Try the statements below, both with and without the decimal points, and note how your results change.

```
In [39]: a = 2.                                # Set up some real variables to play with
In [40]: b = 5.
In [41]: print(a + b)
7.0
In [42]: print(a - b)                         # Negative numbers are displayed as expected
-3.0
In [43]: print(a * b)                         # Multiplication is done with a *
10.0
In [44]: print(a / b)                         # Division is with a forward slash /
0.4
In [45]: print(a ** b)                        # The power operation is done with **
32.0
In [46]: print(b % a)                         # The % operator finds the remainder of a division
1.0
In [47]: print(4.5 % 2)                       # The % operator works with fractional parts too
0.5
```

The above session at the interactive interpreter also illustrates *comments*. This is explanatory text added to programs to help anyone (including yourself!) understand your programs. When Python sees the `#` symbol it ignores the rest of the line. Here we used comments at the interactive interpreter, which is not something one would normally do, as nothing gets saved. When writing modules you should comment your programs comprehensively, though succinctly. They should describe your program in sufficient detail so that someone who is not familiar with the details of the problem but who understands programming (though not necessarily in Python), can understand how your program works. Examples in this manual should demonstrate good practice.

Although you should write comments from the point of view of someone else reading your program, it is in your own interest to do it effectively. You will often come back to read a program you have written some time later, and be surprised by how little of your program you remember. Well written comments will save you a lot of time.

The rules of operator precedence are the same as for most calculators. Python generally evaluates expressions from left to right, but things enclosed in brackets are calculated first, then multiplications and divisions, then additions and subtractions.

```
In [48]: print ( 2 + 3 * 4 )
14
In [49]: print ( 2 + ( 3 * 4 ) )
14
In [50]: print ( ( 2 + 3 ) * 4 )
20
```

Parentheses may also be *nested*, in which case the innermost expressions are evaluated first:

```
In [51]: print ( ( 2 * ( 3 - 1 ) ) * 4 )
16
```

**EXERCISE 3.5:** Play around with the interactive interpreter for a little while until you are sure you understand how Python deals with arithmetic: ensure that you understand the evaluation precedence of operators. Write a program to read the radius of a circle from the keyboard and print its area and circumference to the screen. You need only use an approximate value for  $\pi$  (a good one is 355/113.). Don't forget comments, and make sure the program's output is descriptive, *i.e.* the person running the program is not just left with two numbers but with some explanatory text. Again, think about whether to use `input()` or `raw_input()`.

## 3.6 for loops

### 3.6.1 An example of a for loop

In programming a *loop* is a statement or block of statements that is executed repeatedly. `for` loops are used to do something a fixed number of times. Here is an example:

```
1 sum = 0 # set initial value of sum to zero
2 for i in [0, 1, 2, 3, 4, 5]:
3     print("i now equal to:", i)
4     sum = sum + i**2 # add next element in sum
5     print("sum of squares now equal to:", sum)
6     print("-----")
7
8 print("Done.")
```

The indentation of this program is essential. Copy it into an empty file and name it something like `testloop.py`, which makes it a Python module. The editor will try and help you with the indentation but if it doesn't get it right, indent each line 4 spaces (**N.B.: While it is not strictly forbidden, it is best not to use the TAB key here, because Python gets confused if both spaces and TABs are used to indent**). Finally, **don't forget the colon** at the end of the `for` line. Try and work out what a `for` loop does from the program's output.

The `for` loop is read as a *foreach* loop: "For each item in the list (here, `[0, 1, 2, 3, 4, 5]`'), execute the following indented statements". Remember a list is enclosed in square brackets. On each *iteration* of the loop, the next item in the list is assigned to `i`. In this case that means that the first time around, `i = 0`, the second time `i = 1`, *etc.*

Indentation is an intrinsic part of Python's syntax. In most languages indentation is voluntary and a matter of personal taste. This is not the case in Python. The indented statements that follow the `for` line are called a *nested block*. Python understands the nested block to be the section of code to be repeated by the `for` loop.



Note that the `print("Done.")` statement is not indented. This means it will not be repeated each time the nested block is. The blank line after the nested block is not strictly necessary, but is encouraged as it aids readability.

The editor and interactive interpreter will try and indent for you. If you are using the interpreter it will keep giving you indented lines until you leave an empty line. Then press `Return` or `Enter` again and the loop will be executed.

### 3.6.2 Using the range function

Often we want to do something a large number of times, in which case typing all the numbers becomes tedious. The `range()` function returns a list of numbers, allowing us to avoid typing them ourselves.

You can give `range()` one, two, or three *parameters*,<sup>4</sup> all of which must be integers. If you give `range` one integer, then it produces a list from zero up to one less than that number. **Note that the number itself is not in the range but numbers up to this number are.** This may seem strange but there are reasons.

By way of example, we could have replicated the function of the `for` loop above (*An example of a for loop*) using `range(6)` since:

```
In [52]: print(range(6))
[0, 1, 2, 3, 4, 5]
```

Note that `range(6)` produces a list containing 6 elements. So we could have constructed the `for` loop as follows:

```
1 for i in range(6):
2     print "i now equal to:", i
3     sum += i**2
4     print("sum of squares now equal to:", sum)
5     print("-----")
6 print "Done."
```

In addition to using the `range()` function, note also the use of the `+=` in the `sum` line replacing the longer, but completely equivalent code in the previous version.

Here is another example of the use of `range()` using two input parameters:

```
In [52]: print(range(5, 10))
[5, 6, 7, 8, 9]
```

If you are uncertain what parameters to give `range()` to get it to execute your `for` loop the required number of times, just find the difference between the first and second parameters. This is then the number of times the contents of the `for` loop will be executed.

You may also specify the step, i.e. the difference between successive items. As we have seen above, if you do not specify the step the default is one.

```
In [53]: print(range(10, 20, 2))
[10, 12, 14, 16, 18]
```

Note that this goes up to 18, not 19, as the step is 2. Please keep in mind that the parameters input to `range()` must be integers.

You can always get the number of elements the list produced by `range()` will contain (and therefore the number of times the `for` loop will be executed) by finding the difference between the first and second parameters and dividing that by the step. For example `range(10, 20, 2)` produces a list with 5 elements since  $(20 - 10)/2 = 5$ .

The `range` function will only produce lists of integers. If you want to do something like print out the numbers from 0 to 2, separated by 0.1 some ingenuity is required:

<sup>4</sup> See Section *Using library functions* for a discussion of parameters: for now it is enough to know that they are the things placed within the parentheses.

```
for i in range(20):
    print(i / 10.)
```

You can write this code directly into the IPython console. IPython will even automatically indent the code in the lines after the `for` statement. Try it out. Note that you will get the results you expect only if you include the decimal in the `print(i / 10.)` function. If you leave out the decimal, Python thinks both `i` and the number `10` are integers, and will therefore report the result as an integer. When doing integer division, Python always truncates the result rather than rounding.

Here is a program that brings together many things we have learned in the previous few sections to generate the first  $N$  elements in a Fibonacci sequence:

```
1  # Creates lists of the first N Fibonacci numbers using
2  # 2 different indexing schemes and prints the results.
3
4  N = input("Please specify number of elements in Fibonacci sequence: ")
5
6  FibList1 = [0, 1]    # seed for 1st Fibonacci sequence
7
8  for i in range(2, N):
9      NextElement = FibList1[i-2] + FibList1[i-1]
10     FibList1 = FibList1 + [NextElement]
11
12 print(FibList1)
13
14 FibList2 = [0, 1]    # seed for 2nd Fibonacci sequence
15
16 for i in range(2, N):
17     NextElement = FibList2[-2] + FibList2[-1]
18     FibList2 = FibList2 + [NextElement]
19
20 print(FibList2)
```

The above code produces the first  $N$  terms in a Fibonacci sequence. Actually, it produces the same Fibonacci sequence twice using two different `for` loops that employ different indexing schemes.

**EXERCISE 3.6.1:** Explain how the different indexing schemes work in the two `for` loops in the above code. Why is `NextElement` written as `[NextElement]` in lines 10 and 18?

**EXERCISE 3.6.2:** Use the `range` function to create a list containing the numbers 4, 8, 12, 16, and 20. Write a program to read a number from the keyboard and then print a “table” (don’t worry about lining things up yet) of the value of  $x^2$  and  $x^3$  from 1 up to the number the user typed, in steps of 2.

## 3.7 if statements

### 3.7.1 An example of an if test

The `if` statement executes a nested code block if a condition is true. Here is an example:

```
age = input("Please enter your age: ")
if age > 40:
    print "Wow! You're really old!"
```

Copy this into an empty module and try to work out how `if` works. Make sure to **include the colon** at the end of the `if` statement and indent as in the example.

What Python sees is “if the variable `age` is a number greater than 40 then print a suitable comment”. As in math the symbol “>” means “greater than”. The general structure of an `if` statement is:

```
if [condition]:
    [statements to execute if condition is true]

[rest of program]
```

**Note:** The text enclosed in square brackets is not meant to be typed. It merely represents some Python code.

Again indentation is important. In the above general form, the indented statements will only be executed if the condition is true but the rest of the program will be executed regardless. Its lack of indentation tells Python that it is nothing to do with the `if` test.

### 3.7.2 Comparison tests and Booleans

The `[condition]` is generally written in the same way as in math. The possibilities are shown below:

Comparison	What it tests
<code>a &lt; b</code>	<code>a</code> is less than <code>b</code>
<code>a &lt;= b</code>	<code>a</code> is less than or equal to <code>b</code>
<code>a &gt; b</code>	<code>a</code> is greater than <code>b</code>
<code>a &gt;= b</code>	<code>a</code> is greater than or equal to <code>b</code>
<code>a == b</code>	<code>a</code> is equal to <code>b</code>
<code>a != b</code>	<code>a</code> is not equal to <code>b</code>
<code>a &lt; b &lt; c</code>	<code>a</code> is less than <code>b</code> , which is less than <code>c</code>

The `==` is not a mistake. One `=` is used for *assignment*, which is different to testing for *equality*, so a different symbol is used. Python will complain if you mix them up (for example by doing `if a = 4`).

It will often be the case that you want to execute a block of code if two or more conditions are simultaneously fulfilled. In some cases this is possible using the expression you should be familiar with from algebra: `a < b < c`. This tests whether `a` is less than `b` *and* `b` is also less than `c`.

Sometimes you will want to do a more complex comparison. This is done using *boolean* operators such as `and` and `or`:

```
1 if x == 10 and y > z:
2     print("Some statements which only get executed if")
3     print("x is equal to 10 AND y is greater than z.")
4 if x == 10 or y > z:
5     print("Some statements which get executed if either")
6     print("x is equal to 10 OR y is greater than z")
```

These comparisons can apply to strings too<sup>5</sup>. The most common way you might use string comparisons is to ask a user a yes/no question<sup>6</sup>:

```
answer = raw_input("Evaluate again? ")

if answer == "y" or answer == "Y" or answer == "yes":
    # Do some more stuff
```

**EXERCISE 3.7:** Write a program to ask for the distance travelled and time take for a journey. If they went faster than some suitably dangerous speed, warn them to go slower next time.

<sup>5</sup> although something like `mystring > "hello"` is not generally a meaningful thing to do.

<sup>6</sup> Note string comparisons are case sensitive so `"y" != "Y"`.

### 3.7.3 else and elif statements

`else` and `elif` statements allow you to test further conditions after the condition tested by the `if` statement and execute alternative statements accordingly. They are an extension to the `if` statement and may only be used in conjunction with it.

If you want to execute some alternative statements if an `if` test fails, then use an `else` statement as follows:

```
if [condition]:
    [Some statements executed only if [condition] is true]
else:
    [Some statements executed only if [condition] is false]

[rest of program]
```

If the first condition is true the indented statements directly below it are executed and Python jumps to `[rest of program]`. Otherwise the nested block below the `else` statement is executed, and then Python proceeds to `[rest of program]`.

The `elif` statement is used to test further conditions if (and only if) the condition tested by the `if` statement fails:

```
1 x = input("Enter a number")
2
3 if 0 <= x <= 10:
4     print "That is between zero and ten inclusive"
5 elif 10 < x < 20:
6     print "That is between ten and twenty"
7 else:
8     print "That is outside the range zero to twenty"
```

**EXERCISE 3.7.3:** Write a program to read in two numbers from the user, and then print them out in order from smallest to largest. Modify the program to print three numbers in order. The code for this program will not be too complicated once it is written, but thinking about the logical steps that must be taken in order to sort the three numbers in the most efficient way possible is not easy. Make notes on paper before writing any Python; perhaps draw some diagrams of the flow of the program as it tests the numbers.

## 3.8 while loops

`while` loops are like `for` loops in that they are used to repeat the nested block following them a number of times. However, the number of times the block is repeated can be variable: the nested block will be repeated as long as a condition is satisfied. At the top of the `while` loop a condition is tested and if true, the loop is executed.

```
while [condition]:
    [statements executed if the condition is true]

[rest of program]
```

Here is a short example:

```
1 i = 1
2 while i < 10:
3     print("i equals:", i)
4     i = i + 1
5
6 print("i is no longer less than ten")
```

Recall the role of indentation. The last line is not executed each time the while condition is satisfied, but rather once it is not, and Python has jumped to [rest of program].

The [condition] used at the top of the while loop is of the same form as those used in if statements: see Section *Comparison tests and Booleans*,

Here is a longer example that uses the % (remainder) operator to return the smallest factor of a number entered by a user:

```

1 print("""
2 This program returns the smallest non-unity
3 factor (except one!) of a number entered by the user
4 """)
5 n = input("number: ")
6 i = 2                      # Start at two -- one is a factor of
7                           # everything
8
9 while (n % i) != 0:        # i.e. as long as the remainder of n / i
10     i = i + 1              # is non-zero, keep incrementing i by 1.
11
12 # once control has passed to the rest of the program we know i is a
13 # factor.
14
15 print("The smallest factor of n is:", i )

```

This program does something new with strings. If you want to print lines of text and control when the next line is started, enclose the string in three double quotes and type away.

This is the most complex program you have seen so far. Make sure you try running it yourself. Convince yourself that it returns sensible answers for small n and try some really big numbers.

**EXERCISE 3.8:** Modify the above example of while so that it tells the user whether or not the number they have entered is a prime. Hint: think about what can be said about the smallest factor if the number is prime. Some really bigprimes you could try with your program are 1,299,709 and 15,485,863. Note that this is far from the most efficient way of computing prime numbers!

## 3.9 Using library functions

Python contains a large *library* of standard functions that can be used for common programming tasks. You can also create your own functions (see Section *Making your own functions*). A function is just some Python code, separate from the rest of the program, that performs a task, often taking one or more inputs and returning one or more outputs. Separating out certain pieces of code into functions has at least two advantages. First, a section of code can be reused without rewriting it every time it is needed. Second, functions can break up a programming problem into a number of smaller conceptual tasks, making the entire problem easier to think about and optimize.

Python has some *built-in* functions, for example `type()` and `range()`, which we have already seen. Built-in functions are available to any Python program.

To use a function we *call* it. To do this you type its name, followed by the required *parameters* enclosed in parentheses. Parameters are sometimes called arguments, and are similar to arguments in mathematics. In math, if we write  $\sin(\pi/4)$ , we are using the sin function with the argument  $\pi/4$ . Functions in computing often need more than one variable to calculate their result. These should be separated by commas, and the order you give them in is important.

Even if a function takes no parameters (you will see examples of such functions later), the parentheses must be included.

Functions in a library are contained in a separate *module*, similar to the modules you have been writing and saving thus far. In order to use a particular module, you must explicitly *import* it. This gives you access to the functions it

contains.

The most useful modules for scientific computing are the NumPy and SciPy libraries, which you gain access to by typing `from numpy import *` or `from scipy import *`, respectively, at the top of your program. NumPy contains most of the mathematical functions you will need for doing numerical work. SciPy contains specialized packages for doing certain kinds of computational work such as numerical integration, Fourier transforms, and many other useful things. For this introduction, NumPy will be sufficient for most of our needs.

For example, suppose you want to calculate the sine and cosine of  $\pi/3$ . You could write a module like this:

```
1 from numpy import *
2 mynumber = pi / 3
3
4 print sin(mynumber)
5 print cos(mynumber)
```

When writing a Python module, you must always import NumPy to gain access to its functions. By contrast, when working interactively in the IPython shell, you can use all the NumPy functions without explicitly importing NumPy. This is because NumPy is automatically imported when the IPython shell is launched. This is a convenient feature of the IPython shell, which makes it a bit easier to test snippets of code.

The NumPy module contains many functions, the most useful of which are listed below.

Function	Description
<code>sqrt(x)</code>	Returns the square root of $x$
<code>exp(x)</code>	Return $e^x$
<code>log(x)</code>	Returns the natural log, i.e. $\ln x$
<code>log10(x)</code>	Returns the log to the base 10 of $x$
<code>sin(x)</code>	Returns the sine of $x$
<code>cos(x)</code>	Return the cosine of $x$
<code>tan(x)</code>	Returns the tangent of $x$
<code>arcsin(x)</code>	Return the arc sine of $x$
<code>arccos(x)</code>	Return the arc cosine of $x$
<code>arctan(x)</code>	Return the arc tangent of $x$
<code>fabs(x)</code>	Return the absolute value, i.e. the modulus, of $x$
<code>round(x)</code>	Rounds a float to its nearest integer
<code>floor(x)</code>	Rounds a float <i>down</i> to its integer

The NumPy library also contains two constants: `pi` =  $\pi$  and `e` =  $e$ . These do not require parentheses (see the above example).

Note the `floor` function always rounds down which can produced unexpected results! For example:

```
In [53]: np.floor(-3.01)
Out[53]: -4.0
```

**EXERCISE 3.9:** Use the NumPy library to write a program to print out the sin and cos of numbers from 0 to  $2\pi$  in intervals of  $\pi/6$ . You will need to use the `range()` function.

### Additional notes on importing libraries

The statement `from numpy import *` imports all the functions that are a part of the basic NumPy library. However, there are other ways of importing functions. Here we give a basic overview of these different ways, primarily so you can understand code written using these different methods.

- `from numpy import sin, cos, exp`: This imports only the functions named in the call, here `sin`, `cos`, `exp`, which means the python program or module will be a little bit smaller. It also alerts the user as to which functions are being used and where they came from.

- `import numpy`: This imports the entire NumPy library. However, when imported this way, any function from the library must be called using the “`numpy.`” prefix. For example `sin(x)` would be called as `numpy.sin(x)`. When you use the `from numpy import ...` syntax, no prefix is required.
- `import numpy as np`: This has the same effect as `import numpy` except that we have now given numpy the nickname `np` meaning that any function from the library must be called using the “`np.`” prefix. For example `sin(x)` would be called as `np.sin(x)`. The Python community has, in fact, settled on a few prefixes for standard packages, which we list here for reference:

```
- import numpy as np
- import scipy as sp
- import matplotlib as mpl
- import matplotlib.pyplot as plt
```

**There is an advantage** to using the `import ... as ...` syntax: it ensures that there are no naming conflicts between different modules. For example, if `numpy` and `matplotlib` both had functions named `spam`, then loading their modules with prefixes would give the the names `np.spam` and `mpl.spam`, respectively, thus removing any conflict between the two functions. It also makes it clear to anyone reading the code that a particular set of code is not a part of core Python but comes from a specific module.

When writing modules, we shall follow the widely accepted practice of importing Python modules using their standard prefixes.

## 3.10 Arrays

Arrays are a data type in Python that are used to represent lists of elements, usually numbers. They can be used to represent what in math or physics we commonly refer to as vectors, matrices, or even tensors. In this section, we will focus on one-dimensional arrays, *i.e.* vectors, but the principles we will learn also apply to multidimensional arrays, *i.e.* matrices and tensors. The elements of an array must all be numbers with the same data type: integers, floats, complex, *etc.* Much of the simplicity, efficiency, and speed of numerical calculations in Python come from using arrays. Arrays also simplify the way we conceptualize and process data so we shall use them often.

Arrays are not a “core” Python data type, like integers, floats, and strings. To have access to the `array` data type, we must *import* it from the NumPy library. In addition to the `array` data type, the NumPy library contains many other useful mathematical functions. Therefore, we add the following line to the start of every program in which arrays are used:

```
import numpy as np
```

One such statement suffices to import all the mathematical functions you need (such as sine, cosine, ...) as well as the array data type.<sup>7</sup>

To create an array you use the `array` function:

```
In [55]: xx = array([1, 5, 6.5, -11])
In [56]: print(xx)
[ 1.   5.   6.5 -11. ]
```

The square brackets within the parentheses are required. You can call an array anything you could call any other variable.

Because all the elements of an array must be the same data type, specifying one number in the array using a decimal point, in this case 6.5, makes all the elements in the array have the float data type. This is reflected in the array that’s printed: all of its elements are floats and printed with decimal points.

<sup>7</sup> We reiterate that within the IPython shell, you do not need to import standard NumPy functions such as `array`

We can extend the box analogy used to describe variables in Section [Variables](#) to arrays. An array is a box too, but within it are smaller, numbered boxes. Those numbers start at zero, and go up in increments of one. See Figure 3.1.

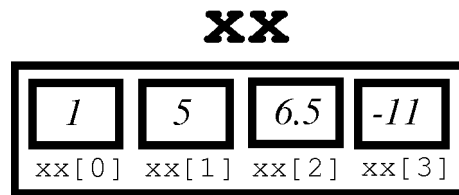


Figure 3.1: Arrays can be thought of as boxes around boxes

This simplifies the program, first of all, by grouping together a set of numbers that are logically related. It also allows the *referencing* of individual elements by *offset*. By referencing we mean either getting the value of an element, or changing it. The first element in the array has the offset [0] (note! *not* 1). The individual element can then be used in calculations like any other float or integer variable. The following example shows the use of referencing by offset using the array created above:

```

In [57]: print(xx)
[ 1.    5.    6.5 -11. ]
In [58]: print(xx[0])
1.0
In [59]: print(xx[3])
-11.0
In [60]: print(range(xx[1])) # Using the element just like any other
[0, 1, 2, 3, 4]             # variable
In [61]: xx[0] = 66.7
In [62]: print(xx)
[ 66.7  5.    6.5 -11. ]
  
```

Let's consider an example. The user has five numbers representing the number of counts made by a Geiger-Müller tube during successive one minute intervals. The following program will read those numbers in from the keyboard, and store them in an array.

```

1 import numpy as np
2
3 counts = np.zeros(5, dtype=int) # See below for an explanation of this
4
5 for i in range(5):
6     print("Minute number", i)
7     response = input("Give the number of counts made in the minute")
8     counts[i] = response
9
10 print("Thank you")
  
```

The contents of the `for` loop are executed five times (see Section [Using the range function](#) if you are unsure). It asks the user for the one minute count each time. Each response is put into the `counts` array, at the offset stored in `i` (which, remember, will run from 0 to 4).

The new thing in that example is the `zeros` function. You cannot get or change the value of an element of an array if that element does not exist. For example, you cannot change the 5th element of a two element array:

```

In [63]: xx = array([3, 4])
In [64]: xx[4] = 99
Traceback (most recent call last):
  File "<pyshell#4>", line 1, in ?
  
```



```
xx[4] = 99
IndexError: index out of bounds
```

Contrast this with numbers (floats and integers) and strings. With these assigning to the variable creates it. With arrays Python must first know how many elements the variable contains so it knows where to put things, *i.e.* “how many boxes are inside the box”.

This means we must create an empty five element array *before* we can start storing the Geiger-Müller counts in it. We could do this by writing `counts = array([0, 0, 0, 0, 0])` but this would quickly get tedious if we wanted a bigger array.

Instead we do it with the `zeros()` function. This takes two parameters, separated by a comma. The first is the number of elements in the array. The second is the type of the elements in the array (remember all the elements are of the same type).

In the Geiger-Müller example we created an array of type `int` because we knew in advance that the number of counts the apparatus would make would necessarily be a whole number. Here are some examples of `zeros()` at work:

```
In [65]: xx = zeros(5, dtype=int)
In [66]: print(xx)
[0 0 0 0 0]
In [67]: yy = zeros(4, dtype=float)
In [68]: print(yy)
[ 0.  0.  0.  0.]
```

### 3.10.1 Operations with arrays

Arrays are not simply lists of numbers in Python but come with a host of useful functions called *methods*. For example, if you want to calculate the sum of all the elements in an array, Python has a simple way of doing that:

```
In [98]: b = array([ 1.,  1.,  2.,  3.,  5.,  8., 13., 21.])
In [99]: b.sum()
Out[99]: 54.0
```

We can calculate the sum of an array simply by attaching “`.sum()`” to the name of the array. (*N.B.* the empty parentheses are necessary!) Arrays come with a number of methods. Here we illustrate a few:

```
In [100]: b.min(), b.max(), b.mean(), b.std(), b.var()
Out[100]: (1.0, 21.0, 6.75, 6.6096520332011428, 43.6875)
```

The last two are the standard deviation and the variance (square of the standard deviation). We can also get the same result by using NumPy functions:

```
In [101]: min(b), max(b), mean(b), std(b), var(b)
Out[101]: (1.0, 21.0, 6.75, 6.6096520332011428, 43.6875)
```

In general, methods execute a little faster than functions, which is reason enough to prefer their use. There is another less apparent advantage. If you import only the array function from NumPy (*e.g.* by typing `from numpy import array`), all the methods come along with array, but the functions, `min`, `max`, `*etc.*` do not. To access them, you must either import the entire NumPy library (`import numpy as np`) or you must import them explicitly from NumPy (`from numpy import min, max, ...`). More documentation on array methods can be found at <http://docs.scipy.org/doc/numpy/reference/generated/numpy.ndarray.html>.

#### Array arithmetic

Two arrays can be added, subtracted, multiplied, and divided using the usual operator notation: `+`, `-`, `*`, and `/`

```

In [43]: c = array(range(1,16,2), dtype=float)
In [44]: b
Out[44]: array([ 1.,  1.,  2.,  3.,  5.,  8., 13., 21.])
In [45]: c
Out[45]: array([ 1.,  3.,  5.,  7.,  9., 11., 13., 15.])
In [46]: b+c
Out[46]: array([ 2.,  4.,  7., 10., 14., 19., 26., 36.])
In [47]: b*c
Out[47]: array([ 1.,  3., 10., 21., 45., 88., 169., 315.])

```

Notice that addition and multiplication are performed element by element of the two arrays. The same is true for division and subtraction as well.

## Arrays and functions

NumPy functions also work on an element-by-element basis. For example `sin(b)` returns an array whose elements are the sines of the elements of `b`. Try it out.

Suppose we want to construct an array to represent  $\sin x \cos x e^{-x^2}$  on the interval from 0 to 5. This is easily done with the following commands.

```

In [48]: x = linspace(0., 5., 100)
In [49]: y = sin(x) * cos(x) * exp(-x**2)

```

The first command creates an array `x` with 100 elements. The next command actually creates three arrays `sin(x)`, `cos(x)`, and `exp(-x**2)`, then multiplies them together on an element-by-element basis to create the desired array `y`.

There are also a number of other useful array functions, including a dot product and cross product that work in the usual way.

```

In [50]: a = array([2., 3., 4.])
In [51]: b = array([-5., 1., -2.])
In [52]: dot(a,b)
Out[53]: -15.0
In [54]: cross(a,b)
Out[55]: array([-10., -16., 17.])

```

You can verify that `cross(b,a)` is equal to `-cross(a,b)` as expected for cross products. The dot and cross products in NumPy are actually much more general and versatile than the above examples might imply, but we will leave that for you to explore.

### 3.10.2 Arrays and loops

Loops are an essential and useful feature of most programming languages, including Python (see §[for loops](#)). However, where we might use loops in languages like C or C++, we often can do the same thing using only arrays in Python. As we shall see, using arrays and avoiding loops is almost always the preferred way of programming in Python, as it results in substantial improvement in speed. It can also make a program easier to read.

**EXERCISE 3.10:** Create two 100-element arrays, one called `a`, consisting of  $\sin(2\pi i/100)$ , and the other called `b`, consisting of  $\cos(2\pi i/100)$ . Find their scalar product  $\mathbf{a} \cdot \mathbf{b}$  defined as

$$\mathbf{a} \cdot \mathbf{b} = \sum_i a_i b_i ,$$

in two different ways. First, calculate the dot product using a `for` loop to calculate the sum, following the example at the beginning of §*An example of a for loop*. You should find that the dot product of the sine and cosine functions is very nearly zero ( $\sim 10^{-15}$ ).<sup>8</sup> Now compute the dot product a more *Pythonic* way by simply multiplying the two arrays together and summing the result using the `sum` method for Python arrays, *i.e.* `(a*b).sum()`. Both ways of calculating the scalar product should give the same answer as they are doing exactly the same calculation.

It turns out the calculating `a * b` by directly multiplying the two arrays and using the `.sum()` method is about 10-20 times faster than calculating the sum using a `for` loop. Code that uses Python's array math is said to be *vectorized*. We have at least two reasons to prefer vectorized code: (1) it is generally much faster than the alternative and (2) it is usually easier to code and read.

Finally we note that it is not always possible to avoid using `for` loops in favor of using vectorized code. We will see some examples later on in this manual. So `for` loops remain a vital part of Python and we will use them often. Nevertheless, when a vectorized version of the code can be written, that is the preferred mode.

## 3.11 Making your own functions

As we saw in Section *Using library functions*, functions are very useful tools for making your programs more concise and modular. Various Python libraries provide a useful range of functions but you will often want or need to write your own functions.

Functions must be *defined* before they are used, so we generally put the definitions at the very top of a program. Here is a very simple example of a function definition that returns the sum of the two numbers that are *passed* to it:

```
In [69]: def addnumbers(x, y):
.....:     sum = x + y
.....:     return sum
.....:

In [70]: x = addnumbers(5, 10)
In [71]: print(x)
15
```

The structure of the definition is as follows:

- The top line must have a `def` statement: this consists of the word `def`, the name of the function, followed by parentheses containing the names of the parameters passed as they will be referred to within the function.[#f9]\_
- Then an indented code block follows. This is what is executed when the function is *called*, *i.e.* used.
- Finally the `return` statement. This is the result the function will return to the program that called it. If your function does not return a result but merely executes some statements then it is not required.

If you change a variable within a function, that change will not be reflected in the rest of the program. For example:

```
In [72]: def addnumbers(x, y):
.....:     sum = x + y
.....:     x = 1000
.....:     return sum

In [73]: x = 5
In [74]: y = 10
In [75]: answer = addnumbers(x, y)
In [76]: print(x, y, answer)
5 10 15
```

<sup>8</sup> For this reason, mathematicians say that sine and cosine are *orthogonal functions*, although instead of doing a sum over many closely spaced points on the interval  $(0, \pi)$ , you perform an integral over the interval  $(0, \pi)$ .

Note that although the variable `x` was changed in the function, that change is not reflected outside the function. This is because the function has its own private set of *local* variables. This is done to minimize the risk of subtle errors in your program.

The relationship between variables used in functions and in the main program is discussed further in Section [Scope](#).

The output of the function above is a single number. It is possible, however, to have virtually any number of outputs of almost any kind. This is done using lists. As an example, the function below outputs the sum, but also outputs a string that tells if the output is an even number. The program below shows different ways the input can be supplied to the function `addnumbers(x, y)`, as well as different ways the output can be retrieved.

Put the following code into a file and name it `AddNum.py`

```

1  def addnumbers(x, y):
2      sum = x + y
3      if sum%2 == 0:
4          evenORnot = "The sum is even"
5      else:
6          evenORnot = "The sum is not even"
7      return sum, evenORnot
8
9  # input numbers and put output into one variable
10 print('*****1*****')
11 ans1 = addnumbers(5, 6)
12 print(ans1)
13 print(ans1[0])
14 print(ans1[1])
15
16 # input numbers and put output into two variables
17 print('*****2*****')
18 resA, resB = addnumbers(5, 6)
19 print(resA)
20 print(resB)
21
22 # input variables with values and put output into one variable
23 print('*****3*****')
24 Anum = 7
25 Bnum = 11
26 ans2 = addnumbers(Bnum, Anum)
27 print(ans2)
28 print(ans2[0])
29 print(ans2[1])
30
31 # input variables with values and put output into two variables
32 print('*****4*****')
33 ans3, ans4 = addnumbers(Bnum, Anum)
34 print(ans3)
35 print(ans4)

```

**EXERCISE 3.11:** Run this program and explain the output of each of the print statements, noting how lists are used in the output of the `addnumbers(x, y)` function.

### 3.11.1 Keyword arguments

Functions can be given arguments that have default values. If you do not want to change the default values, these arguments can be left out of the function call. For example :

```
In [2]: def myfunc(arg1, arg2, kwarg1=1.0, kwarg2=False):
        print(arg1, arg2, kwarg1, kwarg2)
        ...:
        ...:

In [4]: myfunc(2.3, 'zot')
(2.2999999999999998, 'zot', 1.0, False)

In [5]: myfunc(3.1, 5.4, 23)
(3.1000000000000001, 5.4000000000000004, 23, False)

In [6]: myfunc(2.2, 'jack', kwarg2=3)
(2.2000000000000002, 'jack', 1.0, 3)
```

The first two arguments `arg1` and `arg2` are *positional* arguments and must be given input values whenever the function is called. The last two arguments `kwarg1` and `kwarg2` are *keyword* arguments with default values and need not be specified in the call unless values other than the default values are desired. Note that the arguments need not be given in the order defined in the function if the name of the argument is explicitly used in the function call. Try it out with various combinations to test your understanding.

### 3.11.2 Multiple (or no) return values

Functions can have multiple return values or no return values at all if desired. Here is an example of a function that returns two values.

```
def liststats(list):
    return average(list), var(list)
```

Let's try it out. First we define the list code{a} as the input to the function and then we use our function :

```
In [19]: a=[1, 2.3, 3 ,4]
In [20]: liststats(a)
Out[20]: (2.5750000000000002, 1.191875)
```

The function returns a tuple with the two outputs. We can give the outputs individual variable names if we desire :

```
In [21]: mean, var = liststats(a)

In [22]: mean
Out[22]: 2.5750000000000002

In [23]: var
Out[23]: 1.191875
```

A function doesn't have to return any values; it can just perform some task, as in the following example:

```
def square(x):
    print('The square is {0:g}'.format(x*x))
```

Calling this function, we get the desired output:

```
In [20]: square(5)
The square is 25
```

Note that while this function has an output, it does not return a value. For example, if we write :

```
In [21]: a = square(5)
The square is 25
```

we get the desired output. But if we now ask Python to print the value of `a`, we get the following :

```
In [26]: print(a)
None
```

So we see that no value was assigned to `a` by the function. For a function to return a value (or set of values), there must be a `return` function.

The function also uses formatted output, which you can read about at <http://docs.python.org/tutorial/inputoutput.html>.

### 3.11.3 Doc strings

When writing Python functions you are strongly encouraged to document them with a *doc string*. A doc string is a set of comments enclosed by triple quotes that describes what the function does. Such a doc string should appear immediately following the `def` statement in a function definition. Here are a couple of examples of their use:

```
1 def f2c(f):
2     """Converts Fahrenheit to Celsius degrees"""
3     return 5.0*(f-32.0)/9.0
4
5 def SlopeInt(x0, y0, x1, y1):
6     """
7     Calculates the slope m and intercept b for the line
8     y = m x + b that goes through the two points
9     (x0, y0) and (x1, y1).
10
11     Inputs:
12         x0, y0: (x, y) coordinates of a point
13         x1, y1: (x, y) coordinates of another point
14     Returns:
15         m: slope of line
16         b: y-intercept of line
17     """
18     m = (y1-y0)/float(x1-x0)
19     b = y0 - m*x0
20     return m, b
```

The use of docstrings makes it easy to find out what a function does. You can also use the following Python function

```
print(SlopeInt.__doc__)           # note use of double underline characters
```

which returns the docstring

```
Calculates the slope m and intercept b for the line
y = m x + b that goes through the two points
(x0, y0) and (x1, y1).

Inputs:
    x0, y0: (x, y) coordinates of a point
    x1, y1: (x, y) coordinates of another point
Returns:
    m: slope of line
    b: y-intercept of line
```

This syntax works for any function. For example, typing `print(int.__doc__)` returns the docstring for the Python function `int`. This is a nice way to find out what a function does. Try it out.

### 3.11.4 Passing function names to a function

Often we will want to pass the name of a function to another function. This is so easy to do in Python that it's hardly worth mentioning, but because doing so requires special procedures in other languages (*e.g.* Matlab), we show you how to do it explicitly for Python. Suppose we want to find the derivative numerically of an arbitrary function  $f(x)$ . The derivative of  $f(x)$  can be written as

$$f'(x) \approx \frac{f(x+h) - f(x-h)}{2h}. \quad (3.1)$$

Here we write the expression for the derivative such that it is symmetric about  $x$ , which generally gives a more accurate approximation for the derivative at  $x$  when  $h$  remains finite. In the limit  $h \rightarrow 0$ , this gives the same answer as the more traditional asymmetric formula,  $[f(x+h) - f(x)]/h$  favored in math texts. Let's use eq. (3.1) to construct a function to calculate the approximate derivative numerically:

```
def deriv(f, x, h=1.0e-6):
    return (f(x+h)-f(x-h))/(2.0*h)
```

Suppose we want to calculate the numerical derivative of  $e^{-2t}$ . First we write a Python function:

```
def g(t):
    return exp(-2.0*t)
```

Let's find the numerical derivative of  $e^{-2t}$  at  $t = 1$ :

```
In [17]: deriv(g, 1.0)
Out[17]: -0.27067056646012944
```

Let's compare the accuracy of our result to the exact result. The exact derivative of  $e^{-2t}$  is  $-2e^{-2t}$ . Evaluating this at  $t = 1$  gives

```
In [18]: print(-2.0*exp(-2.0*1.0))
-0.270670566473
```

The numerical approximation is correct to the 10<sup>th</sup> decimal place but differs in the 11<sup>th</sup> decimal place – not too shabby! Note also our use of the keyword argument `h` (see §[Keyword arguments](#)).

The above example illustrates how functions can be called as arguments of other functions. As an aside, we shall investigate a little bit further the accuracy of our algorithm for calculating the derivative numerically. To this end, we define another function  $r(x)$  whose derivative we would like to find:  $r(x) = x \sin(\pi x)$ . Again we write the function in Python:

```
def r(x):
    return x*sin(pi*x)
```

Next we write a Python routine that calculates the numerical derivatives of  $g(x)$  and  $r(x)$  numerically at  $x = 1$  and compares them to their analytical counterparts for different values of  $h$ , printing out a table of the results:

```
1 print("\n h          delta g'          delta r'")
2 print("-----")
3 for z in range(1, 18):
4     h = 10.**(-z)
5     x = 1.0
6     gprimeN = deriv(g, x, h)
7     gprimeA = -2.0 * g(x)
8     rprimeN = deriv(r, x, h)
9     rprimeA = sin(pi*x) + x*pi*cos(pi*x)
10    print('{0:5.0e} {1:19.15f} {2:19.15f}'
11          .format(h, gprimeN-gprimeA, rprimeN-rprimeA))
```

We have included some extra formatting commands in the `print` statement so that we get nice columns of numbers in the table Python outputs.<sup>9</sup> Note also that we have split the `print` function into two lines. **In Python you can split any expression contained inside of parentheses between lines** or you can simply split the line anywhere by inserting a backslash “\” character at the end of a line you wish to continue on the next line. Note, however, that the continued line must be indented by at least one space (or as many as you want). Indenting keeps your code neatly contained within the margins. Running the program gives the following output:

```
In [22]: %run deriv.py
```

h	delta g'	delta r'
1e-01	-0.001808082823038	0.051422709840316
1e-02	-0.000018045065330	0.000516745776981
1e-03	-0.000000180447065	0.000005167710426
1e-04	-0.000000001804339	0.000000051676436
1e-05	-0.000000000018823	0.000000000507197
1e-06	0.000000000013096	0.000000000239843
1e-07	0.000000000096363	-0.000000001319586
1e-08	-0.000000000042415	0.000000000900810
1e-09	-0.000000002817973	-0.000000110121494
1e-10	-0.000000002817973	-0.000000998299914
1e-11	-0.000000419151607	0.000016765268480
1e-12	-0.000001806930388	-0.000116461494475
1e-13	0.000053704220843	-0.000338506099399
1e-14	0.000053704220843	-0.011440736345651
1e-15	-0.006885189683064	-0.189076420285677
1e-16	0.131892688395081	0.921146604339480
1e-17	0.270670566473225	3.141592653589793

Notice that the numerical approximation is most accurate for  $h$  in the range  $10^{-5}$  to  $10^{-8}$ . At first glance it is surprising that our approximation gets *worse* as  $h$  gets smaller than  $10^{-8}$ . Can you figure out why? The origin of the problem is that numbers are stored in a computer with a *finite* number of digits, in this case about 15. The two numbers  $f(x + h)$  and  $f(x - h)$  in the numerator of our approximate expression for  $f'$  are very nearly equal for small  $h$ , even to 15 digits. When these two nearly equal numbers are subtracted, the result is a very small number accurate to only a few digits – or less! Python (and most other computer languages) provides ways of doing its calculations with a greater number of digits, but it's usually not necessary. You simply have to be aware of this inherent limitation of computers, and take care to avoid subtracting two nearly equal numbers.

## 3.12 File input and output

Up to this point, all data input and output has been done via the keyboard and screen. However, we often want to read in data from a file on your computer's hard disk or save the results of some calculation to a file for later use. In this section we discuss a few simple methods for reading and writing data files using Python.

### 3.12.1 Reading data from a text file

Suppose you have some data that you have measured that you want to analyze. Being a careful experimenter, you have stored your data you have obtained in a text file named `MyExpt1Data.txt`, which we print out here for reference:

```
Data for falling mass experiment
Date: 3-Sep-2010
Data taken by Lauren and John
```

<sup>9</sup> We won't discuss such formatting in this document. If you want to know more go to <http://docs.python.org/tutorial/inputoutput.html> and read section 7.1.



data point	time (sec)	height (mm)	uncertainty (mm)
0	0	180	3.5
1	0.5	182	4.5
2	1	178	4.0
3	1.5	165	5.5
4	2	160	2.5
5	2.5	148	3.0
6	3	136	2.5
7	3.5	120	3.0
8	4	99	4.0
9	4.5	83	2.5
10	5	55	3.6
11	5.5	35	1.75
12	6	5	0.75

It is critically important that this data file be a *text* file. It cannot be a MSWord file, for example, or an Excel file, or anything other than a plain text file. Such files can be created by a text editor program like **Notepad++** (for a PC) or **TextWrangler** (for a Mac). They can also be created by MSWord and Excel provided you explicitly save the files as text files. **Beware:** You should exit any text file you make and save it with a program that allows you to save the text file using **UNIX**-type formatting, which uses a *line feed* (LF) to end a line. Some programs, like MSWord under Windows, may include a carriage return (CR) character, which can confuse `loadtxt`. Note that we give the file name a *.txt extension*, which indicates to most operating system that this is a *text* file (as opposed to an Excel file, for example, which might have a *.xlsx* or *.xls* extension).

Python provides many functions for reading text files. Here we are going to use a function provided in the NumPy package called `loadtxt`. It has been written to make reading data files particularly simple. For example, to read in the data file `MyExptlData.txt` shown above, you would write:

```
In [83]: dataPoint, time, height, error = loadtxt('MyExptlData.txt',
        skiprows=5, unpack=True)
```

In this case, the `loadtxt` function takes three arguments: the first is a string that is the name of the file to be read, the second tells `loadtxt` to skip the first 5 lines at the top of file (sometimes called the *header*), and the third tells `loadtxt` to output the data (*unpack* the data) so that it can be directly read into arrays. `loadtxt` outputs however many columns of data are present in the text file to the array names listed to the left of the “=” sign. The names labeling the columns in the text file are not used (but you are free to choose the same or similar names, of course, as long as they are legal array names). By the way, for the above `loadtxt` call to work, the file `MyExptlData.txt` should be in the current working directory of the IPython shell. Otherwise, you need to specify the directory path with the file name.

If you don’t want to read in all the columns of data, you can specify which columns to read in using the `usecols` key word. For example, the call

```
In [84]: time, height = loadtxt('MyExptlData.txt',
        skiprows=5, usecols = (1,2), unpack=True)
```

reads in only columns 1 and 2; columns 0 and 3 are skipped. As a consequence, only two array names are included to the left of the “=” sign, corresponding to the two column that are read. Writing `usecols = (0,2,3)` would skip column 1 and read in only the data in columns 0, 2, and 3. In this case, 3 array names would need to be provided on the left hand side of the “=” sign.

One convenient feature of the `loadtxt` function is that it recognizes any *white space* as a column separator: spaces, tabs, *etc.*

Finally you should remember that `loadtxt` is a NumPy function. So if you are using inside a Python module, you must be sure to include an `import numpy` statement before calling `loadtxt`.

### 3.12.2 Writing data to a text file

Of course, you will also need to write or save data to text files. A simple way to write data to a plain text file is to use the Python `open` function:

```
In [85]: fout = open("results.txt", "w")
```

`fout` is a variable like the integers, floats and arrays we have already encountered. It is used to keep track of the particular file that this statement is opening. The `open` function takes two inputs: first a string that is the name of the file to be accessed, and second a *mode*. The possible modes are as follows:

Mode	Description
r	The file is opened for reading
w	The file is opened for writing, and any file with the same name is erased—be careful!
a	The file is opened for appending—data written to it is added on at the end of the file

We won't be using the read keyword (r) here as the `loadtxt` function introduced above is more convenient for reading data files. But we will use the write and append keywords.

Let's say that you want to save the output from some measurements and a calculation to a data file. For example, suppose you have the data arrays `dataPoint`, `time`, `height`, and `error` and you want to write them to a data file like the one shown above. Here is a short Python script that does the job:

```
1 fout = open("output.dat", "w")
2 fout.write("Data for falling mass experiment\n")
3 fout.write("Date: 3-Sep-2010\n")
4 fout.write("Data taken by Lauren and John\n")
5 fout.write("\n")
6 fout.write("data point      time (sec)      height (mm)      uncertainty (mm)\n")
7 for i in range(dataPoint.size):
8     fout.write(str(int(dataPoint[i])) + '\t\t\t\t' +
9               str(time[i]) + '\t\t\t\t' +
10              str(height[i]) + '\t\t\t\t' +
11              str(error[i]) + '\n')
12 fout.close()
```

The first line creates a new data file named `output.dat`, opens it for writing, and assigns it to the variable `fout`. You can use any legitimate variable name of your choice in place of `fout`. The next 5 calls write header information to the file. The `\n` is an “invisible” character that starts a new line. Not including it would make all the text in the various `write` calls appear on the same line. Note that `fout.write("\n")` simply writes a blank line to the data file. Finally, we use a `for` loop to write the data in four columns. The `\t` is another invisible character that writes a tab. Only a single tab is necessary to distinguish columns but here we put in several tabs to better align the data in the columns with headings.

Note that the `write` function takes only a single argument **which must be a string**. Therefore, when writing numerical data to the file, we must convert the numerical variables to strings using the `str` function. Omitting the `str` function results in an error. Notice that different strings are linked together using a “+” sign.

Finally, the data file must be closed using the `close` function. The end result is a file that looks pretty much like the data file introduced towards the beginning of this section.

**EXERCISE 3.12:** Recreate the one hundred element arrays of `sin` and `cos` you created in Exercise 3.10. Print these arrays out to a file in the form of a table. The first line of the table should contain brief descriptive names of the data in each column and the second line should contain first element of both arrays, the third line the second element, and so on. Keep the file as we will use it later.

	A	B	C	D	E
1	Data for falling mass experiment				
2	3-Sep-10				
3	Data taken by Lauren and John				
4	data point	time (s)	height (mm)	uncertainty (mm)	
5	0	0	180	3.5	
6	1	0.5	182	4.5	
7	2	1	178	4	
8	3	1.5	165	5.5	
9	4	2	160	2.5	
10	5	2.5	148	3	
11	6	3	136	2.5	
12	7	3.5	120	3	
13	8	4	99	4	
14	9	4.5	83	2.5	
15	10	5	55	3.6	
16	11	5.5	35	1.75	
17	12	6	5	0.75	

Figure 3.2: Data file in Excel.

### 3.12.3 Reading and writing to CSV files

The Python plotting library `matplotlib`, which we shall encounter later, includes a pair of useful functions for reading and writing to “CSV” or “comma separated value” files. A CSV file is a special kind of text file that spreadsheet programs like Excel can read from and write to. For example, consider the Excel spreadsheet shown in Fig. 3.2. The data and headings in this file can be saved to a CSV file using the Excel *Save As...* menu. The result is a text file with the different columns separated by commas, as shown here:

```
Data for falling mass experiment,,,
3-Sep-10,,,
Data taken by Lauren and John,,,
data point,time (s),height (mm),uncertainty (mm)
0,0,180,3.5
1,0.5,182,4.5
2,1,178,4
3,1.5,165,5.5
4,2,160,2.5
5,2.5,148,3
6,3,136,2.5
7,3.5,120,3
8,4,99,4
9,4.5,83,2.5
10,5,55,3.6
11,5.5,35,1.75
12,6,5,0.75
```

Notice how each row has three commas dividing each row into four columns. This data file and its contents can be read into a Python program using the `matplotlib` function `csv2rec`. The calling sequence is:

```
1 # import the mlab module from matplotlib
2 import matplotlib.mlab as mlab
3 # read data in FallingData.csv into Python
4 r = mlab.csv2rec("FallingData.csv", skiprows=3)
5 # print out the data names (and type) read in from csv file
6 print(r.dtype)
7 # print out the data read in
8 print(r.view)
```

```

9 print(r.data_point)
10 print(r.time_s)
11 print(r.height_mm)
12 print(r.uncertainty_mm)

```

The arguments of the `csv2rec` function are a string containing the file name and, in this case, a second input “`skiprows=3`” that directs Python to skip the first 3 rows when reading the file. Thus the first row read by `csv2rec` is the header row, which it uses to create names for the arrays containing the data in each column. When creating the array names, `csv2rec` changes spaces into underscores and deletes characters, such as parentheses, that cannot be used in array names, and converts all uppercase letters to lowercase. Because this conversion can be a bit confusing, you are well advised to test the conversion by reading the CSV file you are using into a variable (`r` in the above example), and then printing it out, which will reveal the column names that `csv2rec` has created.

The output of the `csv2rec` function call is assigned to a variable, in this case `r`, that is an example of a data structure called a *record array* in NumPy. The record array contains all the data in the columns. The data for each column can be unpacked by combining the record array name, in this case `r`, plus a dot plus the name of the particular column, as shown in the `print` statements in the code example above. Thus, `r.time_s` is an array containing the data in the “time (s)” column from the original Excel file.

It is also possible to write your data to a CSV file. To do so, you first have to pack your data into a NumPy record array and then use a special `matplotlib.mlab` function `rec2csv` to write the data to a CSV file. The code below illustrates how to do this:

```

1 # create some lists of data that you want to save
2 r1 = (1, 2.1, 3.9, 0.2)
3 r2 = (2, 3.3, 4.6, 0.4)
4 r3 = (3, 4.7, 6.8, 0.3)
5 # create a record array
6 rec1 = np.rec.array([r1, r2, r3],
7                     dtype=[('number', int), ('x', float), ('y', float), ('dy', float)])
8 # save record array to a CSV file
9 mlab.rec2csv(rec1, 'testdata.csv')

```

The NumPy function `np.rec.array` packs the data into a record array which, in this case, is called `rec1` (you can choose another name). The first argument of `np.rec.array` is a list of the data arrays. The second argument, which starts with `dtype=` assigns a column name and a data type to each array by means of a list (notice the square brackets `[...]` where the each element in the list is a tuple consisting of a string specifying the column name and a data type (`int` for integer, `float` for floating point number, *etc.*

Finally, the CSV file is created using the `matplotlib.mlab` function `rec2csv`: its first argument is the record array, in this case `rec1`, and the second is a string that is the name of the CSV data file to be created. Note that the data file name has a “`csv`” extension. Opening this file with a spreadsheet program like Excel will produce the expected spreadsheet of data arranged in columns with appropriate column names.

### 3.13 Putting it all together

This section shows a complete, commented program to indicate how most of the ideas discussed so far (variables, arrays, files, *etc.*) are used together.

The program below is a rewritten version of the example introduced in Section *Interpreters, modules, and a more interesting program*, which did nothing more than add two numbers together. However, the two numbers are stored in arrays, the numbers are read in by a separate function, the addition is also done by a separate function, and the result is written to a file.

```

1 '''Writes sum of 2 numbers input by user to a data file'''
2

```

```

3 import numpy as np
4
5 # Declare functions first
6 def addnumbers(x, y):
7     sum = x + y
8     return sum
9
10 def getresponse():
11     # Create a two element array and use it to store the
12     # numbers the user inputs
13     response = np.zeros(2, dtype=float)
14     # Put the first number in the first element of the list:
15     response[0] = input("Please type in a number: ")
16     # Put the second number in the second element:
17     response[1] = input("...and another: ")
18     # And return the array to the rest of the program
19     return response
20
21 # Get name of output file.
22 filename = raw_input("What file would you like to store the result in? ")
23 # Open file for writing:
24 output = open(filename, "w")
25 # Get numbers to add from user and put them into the array numbers
26 numbers = getresponse()
27 # Add the two elements of the array
28 answer = addnumbers(numbers[0], numbers[1])
29 # Convert answer into a string and write string to file
30 stringanswer = str(answer)
31 output.write(stringanswer)
32 # Close the file!
33 output.close()

```

**EXERCISE 3.13:** The following function computes  $e^x$  by summing the Taylor series expansion to  $n$  terms. Explain how the program works by explicitly writing out the calculation for the first 4 terms in the `for` loop. Then write a program to write to a text file a table of,  $x$  and  $e^x$  using both this function and the `exp` function from the `numpy` library, for  $x = 0$  to 1 in steps of 0.1. In addition, write the difference between the two results in a fourth column in the data file. The program should ask the user what value of  $n$  to use.

```

1 def taylor(x, n):
2     sum = 1.0
3     term = 1.0
4     for i in range(1, n):
5         term = term * x / float(i)
6         sum = sum + term
7     return sum

```

# GRAPHICS AND CURVE FITTING

## 4.1 Basic graphical output

Generating graphs is not part of core Python. However, there are a number of Python modules available that do the task quite well. In this course we will use the `matplotlib` module for graphical output. It is a powerful and flexible program that has become the *de facto* standard for 2-d plotting with Python. The `matplotlib` package is vast so we will stick with some fairly simple applications of `matplotlib` for now. These should be adequate for most of your needs in this course.

In any program from which you would like graphical output you should include the lines

```
import numpy as np
import matplotlib.pyplot as plt
```

The plotting module `matplotlib.pyplot` requires `numpy` for almost any significant activity so you should import both modules. The IPython console does this automatically so you do not have to do it *but only if you are operating from within IPython console*. By contrast, it is required in Python modules that you call from the IPython console (or from any other place).

We will demonstrate Python's plotting capabilities by example. To follow what is going on, you need to launch IPython and execute the commands as we go. Instead of writing a program and putting it in a module, we will start by plotting in interactive mode from the IPython console so you can see how each plotting command works. After demonstrating the principal features of Python, we can put it all together in various modules.

Let's say we want to plot the function  $y(x) = x^2$  over the interval from -2 to 2. Here is the code to do it. Type it in to your IPython console.

```
In [3]: x = arange(-2.0, 2.01, 0.05) # creates x array from -2 to 2
                                     # in increments of 0.05
In [4]: y = x**2
In [5]: plot(x, y)
Out[5]: [<matplotlib.lines.Line2D object at 0x1ca36d0>]
```

Here we have introduced a new `numpy` function called `arange`, which does for floating point (real) numbers what the Python function `range` does for integers. Note, however, that `arange` creates an array of numbers while `range` creates a list. When you execute these commands, a new window should appear with a plot of a parabola. The plot that appears extends over a greater range of  $x$  and  $y$  than we might like and it lacks axis labels. Let's fix that:

```
In [6]: xlim(-2, 2)
Out[6]: (-2, 2)

In [7]: ylim(0, 4)
Out[7]: (0, 4)
```

```

In [8]: xlabel('x')
Out[8]: <matplotlib.text.Text object at 0x1ca3ef0>

In [9]: ylabel('y')
Out[9]: <matplotlib.text.Text object at 0x1cb0870>

In [10]: title('Parabola: y(x) = x^2')
Out[10]: <matplotlib.text.Text object at 0x1cc01b0>

```

The meaning of each command should be evident. Your plot should look like that in Fig.4.1. You can save a copy of your plot by clicking on the floppy disk icon at the bottom of the plot window.

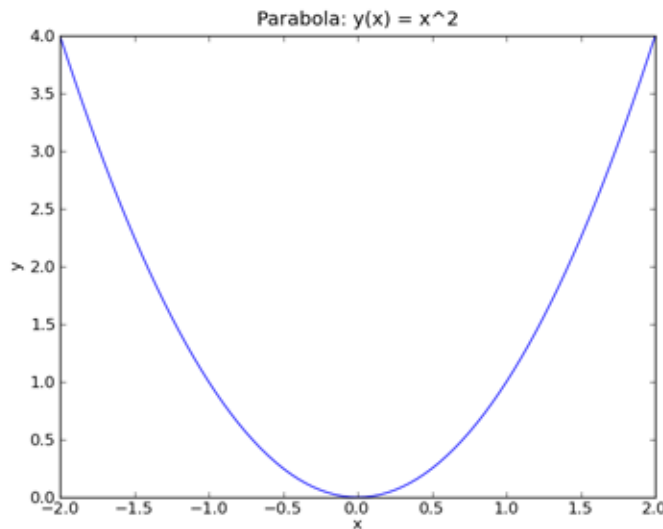


Figure 4.1: Plot of  $y = x^2$  made using `matplotlib.pyplot` Python library

The parabola plot demonstrates only the most basic features of `matplotlib` and while we won't need all of `matplotlib`'s features, we will need a few more than illustrated thus far. It is often convenient, for example, to plot several functions or data sets on one plot. In so doing, we may want to plot symbols, dashed lines, and to give them different colors to distinguish one data set from another. We may also want to plot two plots one above the other. The next example illustrates these capabilities. Please follow along typing each command one after another into the IPython console. This will help you better understand what each command is doing.

### 4.1.1 Subplots

Often you want to create two or more graphs and place them next to one another, generally because they are related to each other in some way. Here we show how to do that, and demonstrate a few more useful plotting tricks in the context of using plots to find graphical solutions to problems where the solutions cannot be found analytically.

There are a number of problems in physics that cannot be solved analytically and thus require numerical solutions. Finding the energy levels for the bound states in a one dimensional square well in quantum mechanics is one such problem. Finding the solutions can be reduced to finding for which real values of  $\theta > 0$  the equation  $\tan \theta = [(8/\theta)^2 - 1]^{1/2}$  is satisfied or the equation  $\cot \theta = -[(8/\theta)^2 - 1]^{1/2}$  is satisfied. You cannot solve this equation analytically, so a graphical solution can be helpful. We would like to plot the two equations on two separate graphs that are stacked one upon the other.

Let's start by defining a function to calculate  $[(8/\theta)^2 - 1]^{1/2}$  :

```
In [11]: def f(t):
.....:     return sqrt((8/t)**2-1)
.....:
```

Note that after you type `def f(t):` and press return, the IPython shell automatically types `.....:`, indicating that it is waiting for more input from you (because your function definition is not finished). It also indents so that the function definition is properly formatted. When you have typed in as many lines as needed, pressing a return will end your function definition and give you another input line.

Next we need to create an array for  $\theta$ . From the equations above, we notice that  $\theta$  cannot be greater than 8 as  $(\theta/8 - 1)^{1/2}$  becomes imaginary for  $\theta > 8$ . Therefore we define a  $\theta$  array that goes from 0 to 8 in steps of 0.1 so that we generate a smooth curve:

```
In [12]: theta = arange(0.0, 8.0, 0.1)
```

Execute each of the following commands and observe how Python creates the plots step by step. Comments are included to help you understand what Python does.

```
In [13]: figure(1)           # creates a box in which the plot will be drawn
Out[13]: <matplotlib.figure.Figure object at 0x1ca3c10>

In [14]: subplot(211)       # 2 => reserve space for 2 rows of plots
                                # 1 => reserve space for 1 column of plots
                                # 1 => set plot 1 (top plot) to be current
                                #      plot and create axes

Out[14]: <matplotlib.axes.AxesSubplot object at 0x1ca3570>
In [15]: plot(theta, tan(theta), 'b', theta, f(theta), 'g')
Out[15]: [<matplotlib.lines.Line2D object at 0x1bae1b0>,
          <matplotlib.lines.Line2D object at 0x1bb54d0>]
```

We have introduced a new function in line [14], the `subplot` function. This function divides the plotting region into a number of smaller regions, called subplots, that can be used to output different plots. In general, the `subplot` function takes three arguments and would be written as `subplot(2, 1, 1)`. However, if all three numbers are less than 10, as they typically are, you can omit the commas, as was done in the code above (this is a very special case that works only for the `subplot` function). The first argument specifies the number of (equally sized) rows and the second specifies the number of (equally sized) columns into which the plotting space is to be divided. The third argument sets the current plotting space, starting from the top left and proceeding left to right and top to bottom. In this case, the plotting space has been divided into 2 rows and 1 column, and the current plotting space has been set to 1, the top row. You will have noticed that the plot you made appears only in the top half of the plotting window.

Input [15] needs a bit more explanation. The `plot` function can plot as many functions or data sets as you wish. The first argument is the  $x$  array and the second is the  $y$  array. These two arrays must have the same number of elements in them (for obvious reasons). The third argument specifies the kind of line that will be drawn and its color. Here we have specified only that line color: `'b'` means blue (the quotes mean it's a string variable). The default is to draw a line so we didn't specify that. The next two arguments specify another  $(x, y)$  pair to be plotted. The next argument `'g'` specifies a green line.

Next we clean up a couple of more items by resetting the plotting limits in the  $y$  direction, drawing a horizontal line at  $y = 0$ , and labeling the  $x$  axis.

```
In [16]: xlabel('x')
Out[16]: <matplotlib.text.Text object at 0x1caf090>

In [17]: ylim(-8, 8)
Out[17]: (-8, 8)

In [18]: axhline()
Out[18]: <matplotlib.lines.Line2D object at 0x1b867f0>
```



We proceed now to create the second plot in the figure. We still need to specify 2 rows and 1 column of subplots for our figure box, but this time we specify that we are working on plot number 2.

```
In [19]: subplot(212)
Out[19]: <matplotlib.axes.AxesSubplot object at 0x1bb5b30>

In [20]: plot(theta, 1/tan(theta), '--r', theta, -f(theta), '-.c')
Out[20]: [<matplotlib.lines.Line2D object at 0x1b90570>,
          <matplotlib.lines.Line2D object at 0x1b90670>]

In [21]: ylim(-8,8)
Out[21]: (-8, 8)

In [22]: axhline()
Out[22]: <matplotlib.lines.Line2D object at 0x1b867f0>
```

We used some extra format specifiers in the format strings in the plot function to show how you can plot with different line styles. If all went well, your plot should look like that shown in Fig. 4.2.

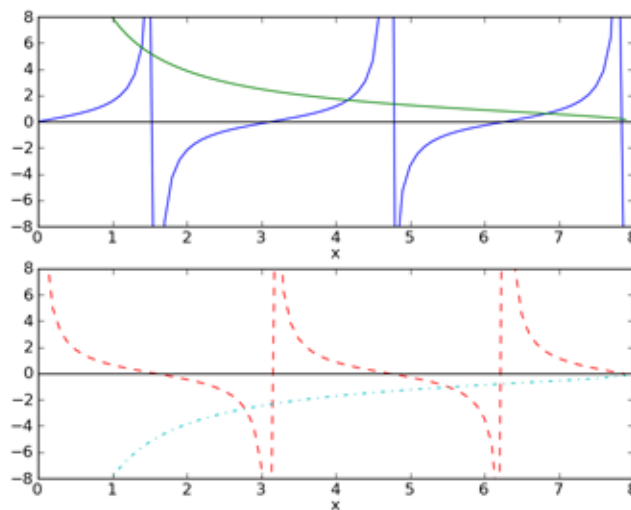


Figure 4.2: Plots demonstrating some of matplotlib's capabilities

We demonstrate a few more of matplotlib's capabilities with the following code. One new feature added here is a legend. To include a legend in your plot, you make a `legend` function call. The only argument you need is the location keyword `loc` which you set equal a string that specifies where the legend will be placed within the plot, *e.g.* `loc='upper left'` or `loc='lower right'` or `loc='best'`, *etc.* (see the matplotlib web site for all the possible location strings). To include a curve in a legend, use the `label` keyword argument in the `plot` function call. Those curves to which you have assigned a label, and those curves only, will be included in the plot legend. Of course, the `legend` function call must come after the `plot` function calls. We save the code for making the graph in in a module (file) that we can execute:

```
1 import matplotlib.pyplot as plt
2 import numpy as np
3
4 def f(x):
5     return np.sin(x) * np.exp(-(x/5.0)**2)
```

```

6
7 xdata = np.linspace(-10., 10., 30)
8 ydata = f(xdata) * (1.0 + 0.3 * np.random.rand(len(xdata)))
9
10 xfit = np.arange(-15., 15.05, 0.1)
11
12 plt.figure(1, figsize = (6,4) )
13 plt.subplot(111)
14 plt.plot(xfit, f(xfit), 'b', label='fit')      # label used in legend
15 plt.plot(xdata, ydata, 'or', label='data')    # label used in legend
16 plt.xlabel('x')
17 plt.ylabel('transverse displacement')
18 plt.legend( loc='upper right' )
19 plt.axhline(color = 'k')
20 plt.axvline(color = 'k')
21
22 plt.savefig('WavyCurve.png')
23
24 plt.show()

```

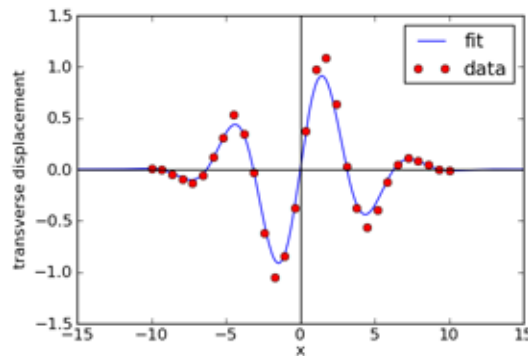


Figure 4.3: Smaller plot with data points, a fit, and a legend.

**Saving a plot to a graphics file.** The function `savefig` in line 22 in the code above saves the plot to a graphics file. The `matplotlib` function `savefig` needs only a string giving the name of the graphics file, including a valid extension, in this case `.png`. The extension `.png` tells `matplotlib` to create a simple bitmapped image, a graphics file type that can be read by any operating system. `matplotlib` recognizes other graphics formats such as `.eps` and `.pdf`.

**EXERCISE 4.1:** The Taylor series for  $\sin(x)$  is  $x - \frac{1}{3!}x^3 + \frac{1}{5!}x^5 - \dots$ . Write a Python routine that plots  $\sin(x)$  and the Taylor expansion of  $\sin(x)$  out to  $n$  terms where  $n$  is the number of terms in the series (for example, the above expansion has 3 terms). When you run your program, it should ask you for the value of  $n$ . Your plot should include axes labels and a legend clearly identifying the two curves. Your plot should extend from  $x = 0$  to  $x = 12$ . Once you get it working, try it out values of  $n$  ranging See EXERCISE 3.13.

## 4.1.2 Error bars

When plotting experimental data it is customary to include error bars that indicate graphically the degree of uncertainty that exists in the determination of each data point. Such uncertainty generally results from the limited resolution with which any physical quantity can be determined. The code below illustrates how `matplotlib` uses the function `errorbar` to add error bars to a plot.

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 # Create x and y data sets
5 x = np.linspace(0.0, 5.0, 25)
6 noise = 0.5*np.random.rand(x.size) # artificial noise
7 y = x*x*(2.0-np.sqrt(x)) + noise
8
9 # Create array of uncertainties (one for each data point)
10 y_unc = 0.5 + 0.1*(0.5*np.random.rand(y.size))
11
12 # Make plot
13 plt.axhline(y=0, color='gray') # draws gray line at y=0
14 plt.plot(x, y, 'ro')
15 plt.errorbar(x, y, yerr=y_unc, ecolor='black', fmt=None)
16
17 # Label axes
18 plt.xlabel('distance (cm)')
19 plt.ylabel('force (N)')
20
21 # save figure
22 plt.savefig('errorbars.png')
23
24 plt.show()

```

The figure below shows the graphical output generated by the above code. The property `ecolor` sets the color of the error bars. Setting the `fmt` property to `None` to means that only the error bars are plotted. Otherwise, `fmt` specifies the format for plotting the symbols. For example, setting `fmt='ro'` would plot solid red data points, making the previous `plot` command unnecessary. Try it out! Just eliminate the `plot` function call and set `fmt='ro'` in the `errorbar` call; you should get the same result as shown in the figure below.

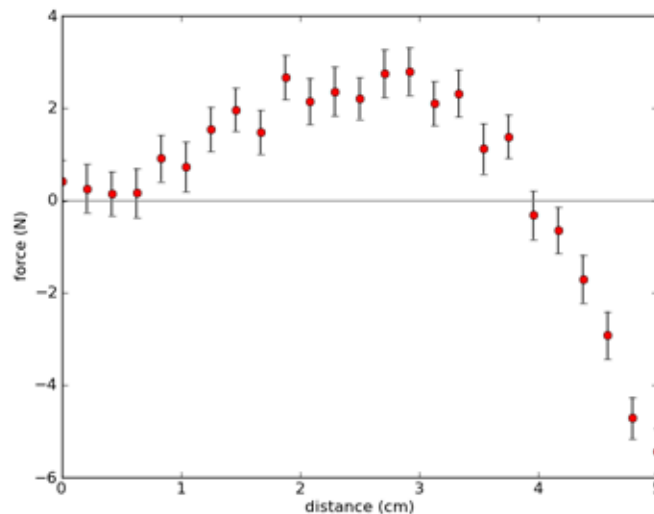


Figure 4.4: Plot of data with error bars.

## 4.2 More advanced graphical output

The plotting methods introduced in the previous section are perfectly adequate for basic plotting and are therefore recommended for simple graphical output. Here, we introduce an alternative syntax that harnesses the full power of `matplotlib`. It gives the user more options and greater control. Perhaps the most efficient way to learn this alternative syntax is to look at an example. Figure 4.4 is produced by the following code:

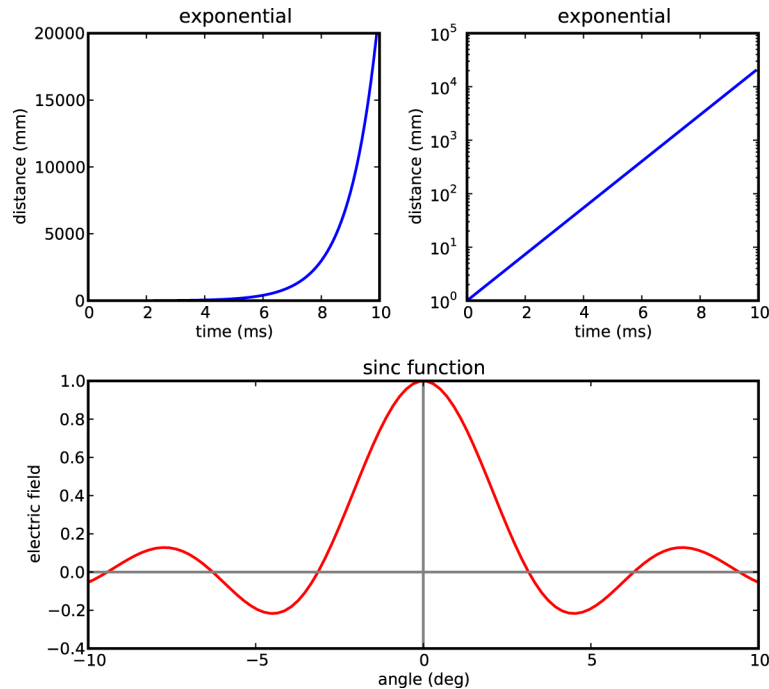


Figure 4.5: Multiple plots in the same window.

```

1  # Demonstrates the following:
2  #     plotting logarithmic axes
3  #     user-defined functions
4  #     "where" function, NumPy array conditional
5
6  import numpy as np
7  import matplotlib.pyplot as plt
8
9  # Define the sinc function, with output for x=0 defined
10 # as a special case to avoid division by zero
11 def s(x):
12     a = np.where(x==0., 1., np.sin(x)/x)
13     return a
14
15 # create arrays for plotting
16 x = np.arange(0., 10., 0.1)
17 y = np.exp(x)
18
19 t = np.linspace(-10., 10., 100)
20 z = s(t)
21
22 # create a figure window
23 fig = plt.figure(1, figsize=(9,8))

```

```

24 # add extra white space between subplots
25 # (needed for axes labels)
26 fig.subplots_adjust(wspace=0.3, hspace=0.3)
27
28
29 # subplot: linear plot of exponential
30 ax1 = fig.add_subplot(2,2,1)
31 ax1.plot(x, y)
32 ax1.set_xlabel('time (ms)')
33 ax1.set_ylabel('distance (mm)')
34 ax1.set_title('exponential')
35
36 # subplot: semi-log plot of exponential
37 ax2 = fig.add_subplot(2,2,2)
38 ax2.plot(x, y)
39 ax2.set_yscale('log')
40 ax2.set_xlabel('time (ms)')
41 ax2.set_ylabel('distance (mm)')
42 ax2.set_title('exponential')
43
44 # subplot: wide subplot of sinc function
45 ax3 = fig.add_subplot(2,1,2)
46 ax3.plot(t, z, 'r')
47 ax3.axhline(color='gray')
48 ax3.axvline(color='gray')
49 ax3.set_xlabel('angle (deg)')
50 ax3.set_ylabel('electric field')
51 ax3.set_title('sinc function')
52
53 plt.savefig('MultiPlotDemo.eps')
54
55 plt.show()

```

After defining several arrays, the above program opens a figure window in line 23 with the statement

```
fig = plt.figure(figsize=(9,8))
```

which is a matplotlib function that opens a blank figure window and assigns it the name (or “handle”) `fig`. Thus, just as we give lists, arrays, and numbers variable names (e.g. `a = [1, 2, 5, 7]`, `dd = np.array([2.3, 5.1, 3.9])`, or `st = 4.3`), we can give a figure window a name; here it is `fig`. In fact we can use the `figure` function to open up multiple windows. The statements

```
fig1 = plt.figure()
fig2 = plt.figure()
```

open up two separate windows, one named `fig1` and the other `fig2`. We can use the names `fig1` and `fig2` to plot things in either window. The `figure` function need not take any arguments or it can take one or more of a number of keyword arguments. In the program listing (line 23), the keyword argument `figsize` sets the width and height of the figure window; the default size is (8, 6); in our program we set it to (9, 8), which is a bit wider and higher than the default size. In the example above, we also choose to open only a single window, hence the single `figure` call.

The `fig.add_subplot(2,2,1)` in line 30 is a matplotlib.pyplot function that divides the figure window into 2 rows (the first argument) and 2 columns (the second argument). The third argument draws a subplot in the first of the 4 subregions (i.e. of the 2 rows  $\times$  2 columns) created by the `fig.add_subplot(2,2,1)` call. To see how this works, type the following code into a Python module and run it:

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 fig = plt.figure(figsize=(9,8))
5 ax1 = fig.add_subplot(2,2,1)
6
7 plt.show()

```

You should get a figure window with axes drawn in the upper left quadrant. The `fig.` prefix used with the `add_subplot(2,2,1)` function directs Python to draw these axes in the figure window named `fig`. If we had opened two figure windows, changing the prefix to correspond to the name of one or the other of the figure windows would direct the axes to be drawn in the appropriate window. Writing `ax1 = fig.add_subplot(2,2,1)` assigns the name `ax1` to the axes in the upper left quadrant of the figure window.

The `ax1.plot(x, y)` in line 31 directs Python to plot the previously-defined `x` and `y` arrays onto the axes named `ax1`.

The `ax2 = fig.add_subplot(2,2,2)` draws axes in the second, or upper right, quadrant of the figure window.

The `ax3 = fig.add_subplot(2,1,2)` draws axes divides the figure window into 2 rows (first argument) and 1 column (second argument). That is, it divides the figure window into 2 halves, top and bottom, and then draws axes in the half number 2 (the third argument), or lower half of the figure window.

The other calls with the prefixes `ax1`, `ax2`, or `ax3`, direct graphical instructions to their respective axes.

The call `ax2.set_yscale('log')` sets the *y*-axes in the second plot to be logarithmic, thus creating a semi-log plot. Creating properly-labeled logarithmic axes like this is more straightforward with the advanced syntax illustrated in the above example.

## 4.3 Curve Fitting

In physics we often have some theoretical curve or *fitting function* that we would like to fit to some experimental data. In general, the fitting function is of the form  $f(x; a, b, c, \dots)$ , where  $x$  is the independent variable and  $a, b, c, \dots$  are parameters to be adjusted so that the function  $f(x; a, b, c, \dots)$  best fits the experimental data. For example, suppose we had some data of the velocity *vs* time for a falling mass. If the mass falls only a short distance such that its velocity remains well below its terminal velocity, we can ignore air resistance. In this case, we expect the acceleration to be constant and the velocity to change linearly in time according to the equation

$$v(t) = v_0 - gt, \quad (4.1)$$

where  $g$  is the local gravitational acceleration. We can fit the data graphically, say by plotting it as shown below in Fig. 4.6 and then drawing a line through the data. When we draw a straight line through a data, we try to minimize the distance between the points and the line, globally averaged over the whole data set.

While this can give a reasonable estimate of the best fit to the data, the procedure is rather *ad hoc*. We would prefer to have a more well-defined analytical method for determining what constitutes a “best fit”. One way to do that is to consider the sum

$$S = \sum_i^n [y_i - f(x_i)]^2, \quad (4.2)$$

where  $y_i$  and  $f(x_i)$  are the values of the experimental data and the fitting function, respectively, at  $x_i$ , and  $S$  is the square of their difference summed over all  $n$  data points. The quantity  $S$  is a sort of global measure of how much the fit  $f(x_i)$  differs from the experimental data  $y_i$ .

Notice that  $S$  is a function only of the fitting parameters  $a, b, \dots$ , since we have summed over all the data points  $x_i$  and  $y_i$ . One way of defining a *best* fit, then, is to find the values of the fitting parameters  $a, b, \dots$  that minimize the  $S$ .

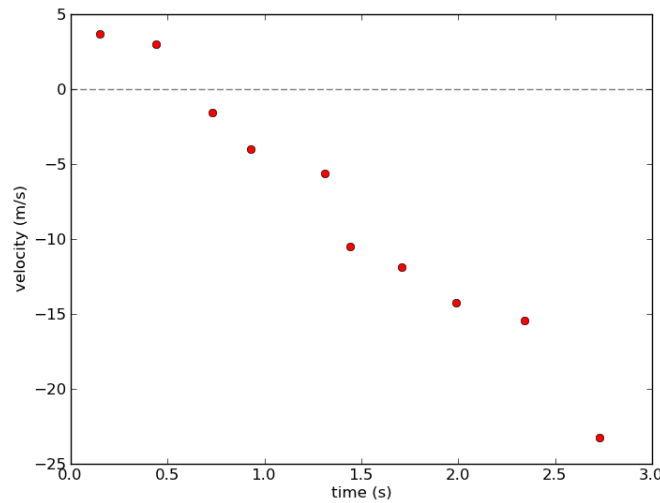


Figure 4.6: Velocity vs time for falling mass.

In principle, finding the values of the fitting parameters  $a, b, \dots$  that minimize the  $S$  is a simple matter. Just set the partial derivatives of  $S$  with respect to the fitting parameter equal to zero and solve the resulting system of equations:

$$\frac{\partial S}{\partial a} = 0, \quad \frac{\partial S}{\partial b} = 0, \dots \quad (4.3)$$

Because there are as many equations as there are fitting parameters, we should be able to solve the system of equations and find the values of the fitting parameters that minimize  $S$ . Solving those systems of equations is straightforward if the fitting function  $f(x; a, b, \dots)$  is linear in the fitting parameters. Some examples of fitting functions linear in the fitting parameters are:

$$\begin{aligned} f(x; a, b) &= a + bx \\ f(x; a, b, c) &= a + bx + cx^2 \\ f(x; a, b, c) &= a \sin x + be^x + ce^{x^2} . \end{aligned} \quad (4.4)$$

For fitting functions such as these, taking the partial derivatives with respect to the fitting parameters, as proposed in (4.3), results in a set of algebraic equations that are linear in the fitting parameters  $a, b, \dots$  and that can thus be solved in a straightforward manner.

For cases in which the fitting function is not linear in the fitting parameters, one can generally still find the values of the fitting parameters that minimize  $S$  but finding them requires more work. We explore a couple of different approaches to the nonlinear case after first developing the methods for the linear case.

### 4.3.1 Linear regression

We start by considering the simplest case, fitting a straight line to a data set, such as the one shown in Fig. 4.6 above. Here the fitting function is  $f(x) = a + bx$ , which is linear in the fitting parameters  $a$  and  $b$ . For a straight line, the sum in (4.2) becomes

$$S(a, b) = \sum_i (y_i - a - bx_i)^2 . \quad (4.5)$$

Finding the best fit in this case corresponds to finding the values of the fitting parameters  $a$  and  $b$  for which  $S(a, b)$  is a minimum. To find the minimum, we set the derivatives of  $S(a, b)$  equal to zero:

$$\begin{aligned}\frac{\partial S}{\partial a} &= \sum_i -2(y_i - a - bx_i) = 2 \left( na + b \sum_i x_i - \sum_i y_i \right) = 0 \\ \frac{\partial S}{\partial b} &= \sum_i -2(y_i - a - bx_i) x_i = 2 \left( \sum_i x_i + b \sum_i x_i^2 - \sum_i x_i y_i \right) = 0\end{aligned}\tag{4.6}$$

Dividing both equations by  $2n$  leads to the equations

$$\begin{aligned}a + b\bar{x} &= \bar{y} \\ a\bar{x} + b\frac{1}{n} \sum_i x_i^2 &= \frac{1}{n} \sum_i x_i y_i\end{aligned}\tag{4.7}$$

where

$$\begin{aligned}\bar{x} &= \frac{1}{n} \sum_i x_i \\ \bar{y} &= \frac{1}{n} \sum_i y_i.\end{aligned}\tag{4.8}$$

Solving Eq. (4.7) for the fitting parameters gives

$$\begin{aligned}b &= \frac{\sum_i x_i y_i - n\bar{x}\bar{y}}{\sum_i x_i^2 - n\bar{x}^2} \\ a &= \bar{y} - b\bar{x}.\end{aligned}\tag{4.9}$$

Noting that  $n\bar{y} = \sum_i y$  and  $n\bar{x} = \sum_i x$ , the results can be written as

$$\begin{aligned}b &= \frac{\sum_i (x_i - \bar{x}) y_i}{\sum_i (x_i - \bar{x}) x_i} \\ a &= \bar{y} - b\bar{x}.\end{aligned}\tag{4.10}$$

While Eqs. (4.9) and (4.10) are equivalent analytically, Eq. (4.10) is preferred for numerical calculations because Eq. (4.10) is less sensitive to roundoff errors. Here is a Python function implementing this algorithm:

```
1 def LineFit(x, y):
2     ''' Returns slope and y-intercept of linear fit to (x,y) data set'''
3     xavg = x.mean()
4     slope = (y*(x-xavg)).sum() / (x*(x-xavg)).sum()
5     yint = y.mean() - slope*xavg
6     return slope, yint
```

It's hard to imagine a simpler implementation of the linear regression algorithm.

**EXERCISE 4.3.1.1:** Write a Python program that creates an artificial noisy data set for a straight line and use the above function to find the best fit to the data (see section 5.8.7 for information on random number generators in Python). Plot the data as unconnected symbols (e.g. circles) and plot the fit as a line (in a single plot).

**EXERCISE 4.3.1.2:** In the above Python function `LineFit(x, y)`, why do we define `xavg = x.mean()`, but do not define `yavg = y.mean()`? Is there any real advantage to doing this?

## 4.3.2 Linear regression with weighting: $\chi^2$

The linear regression routine of the previous section weights all data points equally. That is fine if the absolute uncertainty is the same for all data points. In many cases, however, the uncertainty is different for different points in a



data set. In such cases, we would like to weight the data that has smaller uncertainty more heavily than those data that have greater uncertainty. For this case, there is a standard method of weighting and fitting data that is known as  $\chi^2$  (or *chi-squared*) fitting. In this method we suppose that associated with each  $(x_i, y_i)$  data point is an uncertainty in the value of  $y_i$  of  $\pm\sigma_i$ . In this case, the “best fit” is defined as the one with the set of fitting parameters that minimizes the sum

$$\chi^2 = \sum_i \left( \frac{y_i - f(x_i)}{\sigma_i} \right)^2. \quad (4.11)$$

Setting  $\sigma_i = 1$  for all data points yields the same sum  $S$  we introduced in the previous section. In this case, all data points are weighted equally. However, if  $\sigma_i$  varies from point to point, it is clear that those points with large  $\sigma_i$  contribute less to the sum than those with small  $\sigma_i$ . Thus, data points with large  $\sigma_i$  are weighted less than those with small  $\sigma_i$ .

To fit data to a straight line, we set  $f(x) = a + bx$  and write

$$\chi^2(a, b) = \sum_i \left( \frac{y_i - a - bx_i}{\sigma_i} \right)^2. \quad (4.12)$$

Finding the minimum for  $\chi^2(a, b)$  follows the same procedure used for finding the minimum of  $S(a, b)$  in the previous section. The result is

$$\begin{aligned} b &= \frac{\sum_i (x_i - \hat{x}) y_i / \sigma_i^2}{\sum_i (x_i - \hat{x}) x_i / \sigma_i^2} \\ a &= \hat{y} - b\hat{x}. \end{aligned} \quad (4.13)$$

where

$$\begin{aligned} \hat{x} &= \frac{\sum_i x_i / \sigma_i^2}{\sum_i 1 / \sigma_i^2} \\ \hat{y} &= \frac{\sum_i y_i / \sigma_i^2}{\sum_i 1 / \sigma_i^2}. \end{aligned} \quad (4.14)$$

**EXERCISE 4.3.2.1:** Starting from Eq. (4.12), derive Eqs. (4.13) and (4.14).

For a fit to a straight line, the overall quality of the fit can be measured by the reduced chi-squared parameter

$$\chi_r^2 = \frac{\chi^2}{n - 2} \quad (4.15)$$

where  $\chi^2$  is given by Eq. (4.11) evaluated at the optimal values of  $a$  and  $b$  given by Eq. (4.13). A good fit is characterized by  $\chi_r^2 \approx 1$ . This makes sense because if the uncertainties  $\sigma_i$  have been properly estimated, then  $[y_i - f(x_i)]^2$  should on average be roughly equal to  $\sigma_i^2$ , so that the sum in Eq. (4.11) should consist of  $n$  terms approximately equal to 1. Of course, if there were only 2 terms ( $n=2$ ), then  $\chi^2$  would be zero as the best straight line fit to two points is a perfect fit. That is essentially why  $\chi_r^2$  is normalized using  $n - 2$  instead of  $n$ . If  $\chi_r^2$  is significantly greater than 1, this indicates a poor fit to the fitting function (or an underestimation of the uncertainties  $\sigma_i$ ). If  $\chi_r^2$  is significantly less than 1, then it indicates that the uncertainties were probably overestimated (the fit and fitting function may or may not be good).

We can also get estimates of the uncertainties in our determination of the fitting parameters  $a$  and  $b$ , although deriving the formulas is a bit more involved than we want to get into here. Therefore, we just give the results:

$$\begin{aligned} \sigma_b^2 &= \frac{1}{\sum_i (x_i - \hat{x}) x_i / \sigma_i^2} \\ \sigma_a^2 &= \sigma_b^2 \frac{\sum_i x_i^2 / \sigma_i^2}{\sum_i 1 / \sigma_i^2}. \end{aligned} \quad (4.16)$$

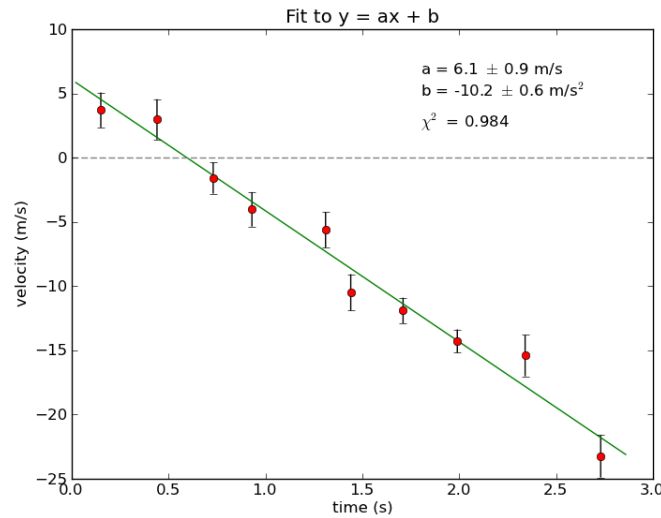


Figure 4.7: Fit using  $\chi^2$  least squares fitting routine with data weighted by error bars.

The estimates of uncertainties in the fitting parameters depend explicitly on  $\{\sigma_i\}$  and will only be meaningful if (i)  $\chi_r^2 \approx 1$  and (ii) the estimates of the uncertainties  $\sigma_i$  are accurate.

You can find more information, including a derivation of Eq. (4.16), in *Data Reduction and Error Analysis for the Physical Sciences*, 3rd ed by P. R. Bevington & D. K. Robinson, McGraw-Hill, New York, 2003.

**EXERCISE 4.3.2.2:** Write a Python function implementing linear regression with  $\chi^2$  weighting, that is, using Eqs. (4.13) and (4.14). Also return the estimates of the uncertainties  $\sigma_a$  and  $\sigma_b$  given by (4.16). Include a warning about the accuracy of the returned values of  $\sigma_a$  and  $\sigma_b$  if  $|\chi_r^2 - 1| > 0.8$ . Test your program by setting all  $\sigma_i$  to a constant  $\sigma$  and redoing Exercise 4.3.1.1. Your output should look something like Fig. 4.6

### 4.3.3 Nonlinear fitting using linear regression with $\chi^2$ weighting

We can use our results for linear regression with  $\chi^2$  weighting to fit functions that are nonlinear in the fitting parameters, provided we can transform the fitting function into one that is linear in the fitting parameters.

To illustrate this approach, let's consider some experimental data taken from a radioactive source that was emitting beta particles (electrons). We notice that the number of electrons emitted per unit time is decreasing with time. Theory suggests that the number of electrons  $N$  emitted per unit time should decay exponentially according to the equation

$$N(t) = N_0 e^{-t/\tau} . \quad (4.17)$$

This equation is nonlinear in the fitting parameter  $\tau$  and thus cannot be fit using the method of the previous

We begin our linear regression analysis by transforming our fitting function to have a linear form. To this end we take the logarithm of Eq. (4.17) :

$$\ln N = -\frac{t}{\tau} + \ln N_0 .$$

With this transformation our fitting function is linear in the independent variable  $t$ . To make our method work, we only have to redefine the dependent variable and fitting parameters. In this case, this means

$$\begin{aligned} y_i &= \ln N_i \\ a &= \ln N_0 \\ b &= -1/\tau \end{aligned} \quad (4.18)$$

Of course, an *essential* part of this transformation is that we also have to transform the estimates of the uncertainties  $\delta N_i$  in  $N_i$  to the uncertainties  $\delta y_i$  in  $y_i (= \ln N_i)$ . So how much does a given uncertainty in  $N_i$  translate into an uncertainty in  $y_i$ ? In most cases, the uncertainty in  $y_i$  is much smaller than  $y_i$ , *i.e.*  $\delta y_i \ll y_i$ ; similarly  $\delta N_i \ll N_i$ . In this limit we can use differentials to figure out the relationship between these uncertainties. Here is how it works for this example:

$$\begin{aligned} y_i &= \ln N_i \\ \Rightarrow dy_i &= \frac{1}{N_i} dN_i \\ \Rightarrow \delta y_i &= \frac{\delta N_i}{N_i} . \end{aligned} \tag{4.19}$$

Here we identify the differentials  $dy_i$  and  $dN_i$  with the uncertainties  $\delta y_i$  and  $\delta N_i$ .

The only remaining task is to estimate the uncertainties in  $N_i$ . Here we suppose that the uncertainty  $\delta N_i$  comes from simple counting statistics. In this case  $\delta N_i \approx \sqrt{N_i}$ . Thus from Eq. (4.19)

$$\delta y_i = \frac{1}{\sqrt{N_i}} \tag{4.20}$$

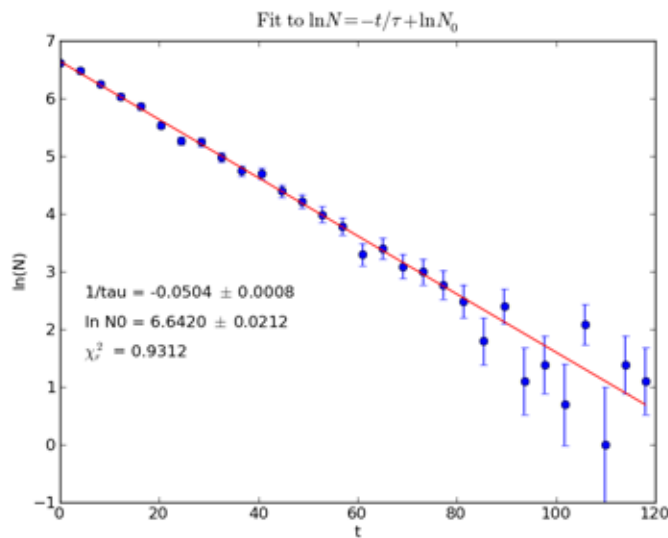


Figure 4.8: Semi-log plot of beta decay measurements from Phosphorus-32.

Fig. 4.7 shows a fit to artificially generated beta decay data. The Python code for generating fig. 4.7 is provided below. Note that the error bars are large when the number of counts  $N$  are small. This is consistent with what is known as *shot noise* (noise that arises from counting discrete events), which obeys *Poisson* statistics. You will study sources of noise, including shot noise, later in your lab courses.

```

1 # Routine FitLinWtsErrBars
2 import numpy as np
3 import numpy.random
4 import matplotlib.pyplot as plt
5 import LinearFitting # from Exercise 4.3.2.2
6
7 n = 30
8

```

```

9  # Create raw data
10 tau = 20.2      # Phosphorus-32 half life = 14 days; tau = t_half/ln(2)
11 N0 = 750.       # Initial count rate (per second)
12
13 t = np.linspace(0., 118.0, n)      # n measurements over 118 days
14 N = N0*np.exp(-t/tau)              # Decay signal w/o noise
15 for i in range(n):                 # make a noisy signal
16     N[i] = numpy.random.poisson(N[i], 1) # Poisson statistics for shot noise
17     sigN = np.sqrt(N)               # Estimate of uncertainties for raw data
18
19 # Take np.log to transform data to linear form
20 y = np.log(N)                      # transform data for fitting
21 sigy = sigN/N                      # transform uncertainties for fitting
22
23 yint, slope, redchisqr, sigyint, sigslope = LinearFitting.LineFitWt(t, y, sigy)
24 tfit = np.array([t.min(), t.max()])
25 yfit = slope*tfit + yint
26
27 plt.errorbar(t, y, sigy, fmt='bo')
28 plt.plot(tfit, yfit, 'r-')
29 plt.title('$\mathrm{Fit}$ to $\ln N = -t/\tau + \ln N_0$')
30 plt.xlabel('t')
31 plt.ylabel('ln(N)')
32 plt.text(5., 2.5, '1/tau = {0:6.4f} $\pm$ {1:6.4f}'.format(slope, sigslope))
33 plt.text(5., 2.0, 'ln N0 = {0:6.4f} $\pm$ {1:6.4f}'.format(yint, sigyint))
34 plt.text(5., 1.4, '$\chi_r^2 = {0:6.4f}'.format(redchisqr))
35 plt.show()

```

**EXERCISE 4.3.3.1:** The output of the above Python routine gives the values of  $1/\tau$  and  $\ln N_0$  and their respective uncertainties. What are the values of  $\tau$  and  $N_0$  and their respective uncertainties?

### 4.3.4 Nonlinear fitting

The method introduced in the previous section for fitting nonlinear fitting functions can be used only if the fitting function can be transformed into a linear fitting function. When we have a nonlinear fitting function that cannot be transformed into a linear form, we need another approach.

The problem of finding values of the fitting parameters that minimize  $\chi^2$  is a nonlinear optimization problem to which there is quite generally no analytical solution (in contrast to the linear optimization problem). We can get a better idea about this nonlinear optimization problem, namely fitting a nonlinear fitting function to a data set, by considering a fitting function with only two fitting parameters. That is, we are trying to fit some data set  $\{x_i, y_i\}$ , with uncertainties in  $\{y_i\}$  of  $\{\sigma_i\}$ , to a fitting function is  $f(x; a, b)$  where  $a$  and  $b$  are the two fitting parameters. We are trying to find the minimum in

$$\chi^2(a, b) = \sum_i \left( \frac{y_i - f(x_i)}{\sigma_i} \right)^2.$$

Note that once the data set, uncertainties, and fitting function are specified,  $\chi^2(a, b)$  is simply a function of  $a$  and  $b$ . We can picture the function  $\chi^2(a, b)$  as a landscape with peaks and valleys: as we vary  $a$  and  $b$ ,  $\chi^2(a, b)$  rises and falls. The basic idea of all nonlinear fitting routines is to start with some initial guesses for the fitting parameters, here  $a$  and  $b$ , and by scanning the  $\chi^2(a, b)$  landscape, find values of  $a$  and  $b$  that minimize  $\chi^2(a, b)$ .

There are a number of different methods for trying to find the minimum in  $\chi^2$  for nonlinear fitting problems. Nevertheless, the method that is most widely used goes by the name of the *Levenberg-Marquardt* method. Actually the Levenberg-Marquardt method is a combination of two other methods, the *steepest descent* (or gradient) method and *parabolic extrapolation*. Roughly speaking, when the values of  $a$  and  $b$  are not too near their optimal values, the gradient descent method determines in which direction in  $(a, b)$ -space the function  $\chi^2(a, b)$  decreases most quickly—the

direction of steepest descent—and then changes  $a$  and  $b$  accordingly to move in that direction. This method is very efficient unless  $a$  and  $b$  are near their optimal values. Thus, as this occurs, the Levenberg-Marquardt method gradually changes to the parabolic extrapolation method, which approximates  $\chi^2(a, b)$  by a Taylor series second order in  $a$  and  $b$  and then computes directly the analytical minimum of Taylor series approximation of  $\chi^2(a, b)$ . This method is only good if the second order Taylor series provides a good approximation of  $\chi^2(a, b)$ . That is why parabolic extrapolation only works well near the minimum in  $\chi^2(a, b)$ .

Before illustrating the Levenberg-Marquardt method, we make one important cautionary remark: the Levenberg-Marquardt method can fail if the initial guesses of the fitting parameters are too far away from the desired solution. This problem becomes more serious the greater the number of fitting parameters. Thus it can be important to provide reasonable initial guesses for the fitting parameters. Usually, this is not a problem, as it is clear from the physical situation of a particular experiment what reasonable values of the fitting parameters are. But beware!

The `scipy.optimize` module provides routines that implement the Levenberg-Marquardt non-linear fitting method. One is called `scipy.optimize.leastsq`. A somewhat more user-friendly version of the same method is accessed through another routine in the same `scipy.optimize` module: it's called `scipy.optimize.curve_fit` and it is the one we shall demonstrate here. The function call is

```
import scipy.optimize
[... insert code here ...]
scipy.optimize.curve_fit(f, xdata, ydata, p0=None, sigma=None, **kwargs)
```

The arguments of `curve_fit` are

- `f(xdata, a, b, ...)`: is the fitting function where `xdata` is the data for the independent variable and `a, b, ...` are the fitting parameters, however many there are, listed as separate arguments. Obviously, `f(xdata, a, b, ...)` should return the  $y$  value of the fitting function.
- `xdata`: is the array containing the  $x$  data.
- `ydata`: is the array containing the  $y$  data.
- `p0`: is a tuple containing the initial guesses for the fitting parameters. The guesses for the fitting parameters are set equal to 1 if they are left unspecified.
- `sigma`: is the array containing the uncertainties in the  $y$  data.
- `**kwargs`: are keyword arguments that can be passed to the fitting routine `scipy.optimize.leastsq` that `curve_fit` calls. These are usually left unspecified.

We demonstrate the use of `curve_fit` to fit artificially generated data to a power law  $d(t) = d_0 t^{-\nu}$ . The data consists of measurements of a correlation length  $d$  measured at different reduced temperatures  $t$ . The Python code is given below and the output is plotted in Fig. 4.8.

```
1  '''
2  This routine demonstrates nonlinear fitting using two different approaches:
3  (1) Levenburg-Marquardt algorithm
4  (2) linear fit to power law data transformed to linear form
5  The fitting function is a power law (modeling beta decay).
6  Artificial data is created and both methods fit the same data, which is
7  then plotted in separate plots that also report the results of the fits.
8
9  '''
10 import numpy as np
11 import numpy.random
12 import matplotlib.pyplot as plt
13 import scipy.optimize
14 import LinearFitting          # from Exercise 4.3.2.2
15
16 n = 30
17
```

```

18 # Create raw data
19 nu = 1.0/3.0                # exponent for correlation length
20 d0 = 10.                    # power law amplitude
21
22 t = np.logspace(-2.8, -0.3, n) # n measurements
23 d = d0*t**(-nu)              # Decay sigdal w/o noise
24 for i in range(n):          # make a noisy sigdal
25     d[i] = numpy.random.poisson(d[i], 1) # use Poisson statistics
26 sigd = np.sqrt(d)           # Estimate of uncertainties for raw data
27
28 # Perform nonlinear fit
29 def f(t, d0, nu):           # Nonlinear fitting function
30     return d0*t**(-nu)
31
32 nlfitt, nlpccov = scipy.optimize.curve_fit(f, t, d, p0=[d0, nu], sigma=sigd)
33 # Give outputs of fit pretty names
34 d0_nlfitt = nlfitt[0]
35 nu_nlfitt = nlfitt[1]
36 d0sig_nlfitt = np.sqrt(nlpccov[0][0])
37 nusig_nlfitt = np.sqrt(nlpccov[1][1])
38
39 tfit = np.array([t.min(), t.max()])
40 yfitNL = f(tfit, d0_nlfitt, nu_nlfitt)
41
42 # Computer reduced chi square for nonlinear fit
43 redchisqrNL = (((d - f(t, d0_nlfitt, nu_nlfitt))/sigd)**2).sum()/float(len(t)-2)
44
45 # Take np.log to transform data to linear form
46 x = np.log(t)
47 y = np.log(d)              # transform data for fitting
48 sigy = sigd/d              # transform uncertainties for fitting
49
50 # Do linear fit to transformed data
51 yint, slope, redchisqr, sigyint, sigslope = LinearFitting.LineFitWt(x, y, sigy)
52 yfit = slope*np.log(tfit) + yint
53
54 # Figure code to make two plots in one figure follows
55 fig = plt.figure(figsize=(15, 5))
56
57 # Plot results for nonlinear fit
58 ax1 = fig.add_subplot(121)
59 ax1.plot(tfit, yfitNL, 'r-')
60 ax1.errorbar(t, d, sigd, fmt='bo')
61 ax1.set_title('$\mathrm{Nonlinear}\ \mathrm{fit}\ \mathrm{to}\ d(t) = d_0\ t^{-\nu}$')
62 ax1.set_xscale('log')
63 ax1.set_yscale('log')
64 ax1.set_xlabel('t (reduced temperature)')
65 ax1.set_ylabel('d (nm)')
66 ax1.set_ylim(10., 100.)
67 ax1.text(0.95, 0.90,
68         '$\nu = {0:6.3f}\ \mathrm{pm}\ {1:6.3f}$'.format(nu_nlfitt, nusig_nlfitt),
69         ha='right', va='bottom', transform = ax1.transAxes)
70 ax1.text(0.95, 0.83,
71         '$d_0 = {0:6.1f}\ \mathrm{pm}\ {1:6.1f}$'.format(d0_nlfitt, d0sig_nlfitt),
72         ha='right', va='bottom', transform = ax1.transAxes)
73 ax1.text(0.95, 0.76,
74         '$\chi_r^2 = {0:6.4f}$'.format(redchisqrNL),
75         ha='right', va='bottom', transform = ax1.transAxes)

```

```

76
77 # Plot results for linear fit
78 ax2 = fig.add_subplot(122)
79 ax2.plot(tfit, np.exp(yfit), 'r-')
80 ax2.errorbar(t, d, sigd, fmt='bo')
81 ax2.set_title('$\mathrm{Linear}\ \mathrm{fit}\ \mathrm{to}\ \mathrm{ } \ln\, d(t) = -\nu \ln\, t + \ln\, d_0$')
82 ax2.set_xscale('log')
83 ax2.set_yscale('log')
84 ax2.set_xlabel('t (reduced temperature)')
85 ax2.set_ylabel('d (nm)')
86 ax2.set_ylim(10., 100.)
87 ax2.text(0.95, 0.90,
88         '$\nu = {0:6.3f} \pm {1:6.3f}$'.format(-slope, sigslope),
89         ha='right', va='bottom', transform = ax2.transAxes)
90 ax2.text(0.95, 0.83,
91         '$\ln d_0 = {0:6.4f} \pm {1:6.4f}$'.format(yint, sigyint),
92         ha='right', va='bottom', transform = ax2.transAxes)
93 ax2.text(0.95, 0.76,
94         '$\chi_r^2 = {0:6.4f}$'.format(redchisqr),
95         ha='right', va='bottom', transform = ax2.transAxes)
96 ax2.text(0.95, 0.65,
97         '$\nu = {0:6.3f} \pm {1:6.3f}$'.format(-slope, sigslope),
98         ha='right', va='bottom', transform = ax2.transAxes)
99 ax2.text(0.95, 0.58,
100         '$d_0 = {0:6.1f} \pm {1:6.1f}$'.format(np.exp(yint), sigyint*np.exp(yint)),
101         ha='right', va='bottom', transform = ax2.transAxes)
102 plt.savefig('FitPower.png')
103 plt.show()

```

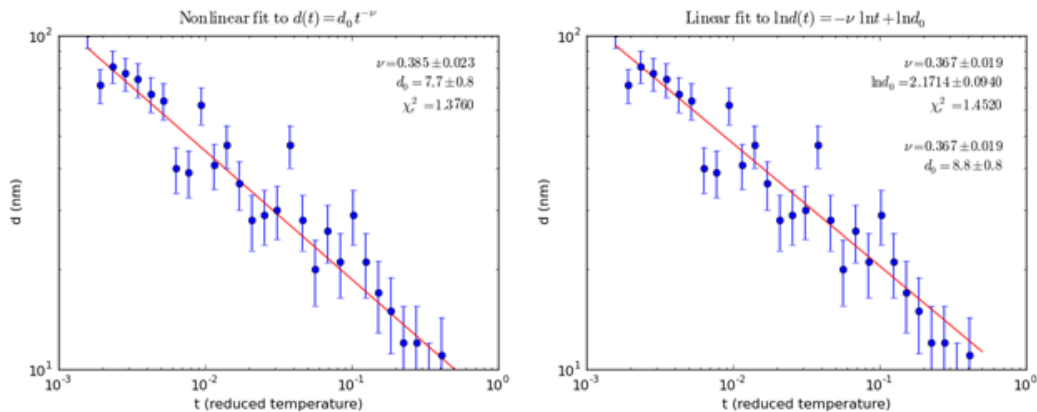


Figure 4.9: Fits to power law data [Left] using the Levenburg-Marquardt algorithm for nonlinear least squares fitting and [Right] using linear least squares to fit data transformed to linear form.

# ADDITIONAL PYTHON

## 5.1 Running Python in other shells

### 5.1.1 Making your Python program callable from a Linux or Unix shell

The first line of any Python module that you might conceivably want to run from a Linux or Unix shell (this includes MacOS X) should be:

```
#!/Library/Frameworks/Python.framework/Versions/Current/bin/python
```

This line is only executed if the file is called from a Linux or Unix shell. Because it starts with the Python comment symbol #, this line is ignored when the module is run from any Python shell (including Pylab).

Note that this is only true for the Enthought distribution of Python. For other distributions, you might need one of the following

```
#!/usr/local/bin/python
#!/usr/bin/python
```

### 5.1.2 Making your Python program callable from a Windows

If you plan to run your module from a Windows machine, but outside a Python shell, you should include a command like this at the top of your program

```
#! C:\???
```

## 5.2 List Comprehensions

*List comprehensions* provide a concise convenient way for creating lists. They are best explained by a few demonstrations :

```
In [1]: a = [i for i in range(5)]
In [2]: a
Out[2]: [0, 1, 2, 3, 4]
In [3]: b = [x**(3./2.) for x in [4, 9, 16, 25]]
In [4]: b
Out[4]: [8.0, 27.0, 64.0, 125.0]
```

A list comprehension consists of an expression followed by a `for` clause. The `for` clause can itself have more `for` and/or `if` clauses embedded in it. For example:



```
In [5]: c = [r for r in b if r>30]
In [6]: c
Out[6]: [64.0, 125.0]
```

Notice how `r` is just a dummy variable that refers to any element in the list `b`. List comprehensions work on any kind of list, including lists of strings:

```
In [7]: names = ['Patrick', 'Sarah', 'Stephen', 'Joyce', 'Nina', 'Andrew']
In [8]: N_names = [x for x in names if x[0]=='N']
In [9]: N_names
Out[9]: ['Nina']
```

## 5.3 Arrays in Python

In lower level languages common mathematical operations on arrays must be done “manually”. For example, we might have a three element array that represents a vector. To double the length of the vector we simply multiply it by two:

$$2 \times \begin{pmatrix} 2 \\ 4 \\ -11 \end{pmatrix} = \begin{pmatrix} 4 \\ 8 \\ -22 \end{pmatrix}$$

In many languages (C for example) the programming equivalent is more complicated. You step through the array an element at a time, multiplying each element by two. In Python this might be done like this:

```
In [1]: xx = array([2, 4, -11])
In [2]: yy = zeros(3, dtype=int) # Create empty array ready to receive result
In [3]: for i in range(0, 3):
        yy[i] = xx[i] * 2
In [4]: print(yy)
[ 4  8 -22]
```

However, this is not required. Python’s arrays “understand” common mathematical operations and will generally do the right thing. Examples follow.

If you add a number to an array, it gets added to each element...

```
In [5]: xx = array([2, 4, -11])
In [6]: yy = xx + 0.1
In [7]: print(yy)
[ 2.1  4.1 -10.9]
```

...and if you multiply an array by a number, each element gets individually multiplied (like vectors)...

```
In [8]: print(xx * 2)
[ 4  8 -22 ]
```

If you add two arrays together the corresponding element in each array gets added together (again, like vectors)...

```
In [9]: zz = array([5, 5, 5])
In [10]: print(xx + zz)
[ 7  9 -6]
```

...but of course the arrays must be of the same dimensions (see Figure *Arrays must be of the same dimensions in order to add them together*):

```
In [11]: zz = array([5, 5, 5, 5])
In [12]: print(xx + zz)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ValueError: frames are not aligned
```

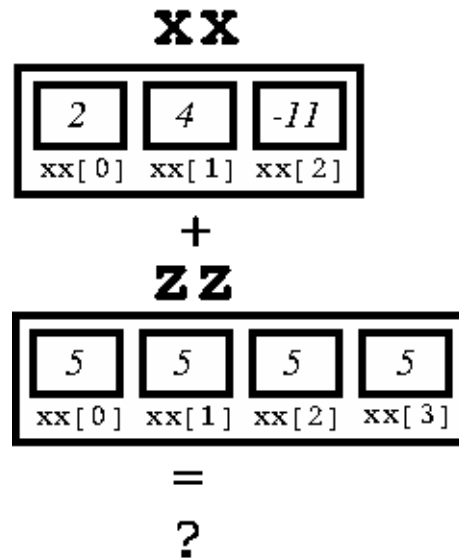


Figure 5.1: Arrays must be of the same dimensions in order to add them together

These kind of statements save a lot of time. The array library contains many other useful functions. For example, one can compute the dot product of two arrays using the `dot()` function. This provides a much quicker way of doing the exercise in Section [Arrays](#).

Arrays can be of more than one dimension. A two-dimensional array is similar to a matrix in mathematics. Consider the following matrix:

$$\begin{pmatrix} 5.3 & -10 \\ 4 & 16 \end{pmatrix}$$

We create such an array in Python as follows:

```
In [13]: xx = zeros([2,2], dtype=float) # The 2nd entry (dtype=float) function
                                             # in zeros is not necessary here since
                                             # the default is float

In [14]: print(xx)
[[ 0.,  0.],
 [ 0.,  0.]]
In [15]: xx[0][0] = 5.3
In [16]: print(xx)
[[ 5.3  0. ]
 [ 0.   0. ]]
In [17]: xx[1][0] = 4
In [18]: xx[0][1] = -10
In [19]: xx[1][1] = 16
In [20]: print(xx)
[[ 5.3 -10. ]
 [ 4.   16. ]]
In [21]: print(xx[1][0])
4.0
```

The first parameter that `zeros()` is passed maybe a list rather than a single number. This then gives the dimensions of the array (number of rows first, then columns). Elements of the array are then indexed using *two* numbers enclosed in square brackets. First the row offset from the top left, then the column offset.

Two-dimensional arrays are used widely in Physics, and also in the manipulation of graphical images.

## 5.4 Functions you may need

### 5.4.1 Random numbers

Some of the problems require random numbers. NumPy contains a random number library under `numpy.random` with a broad range of random number generators. The simplest is `random.rand()`, which generates a random number sampled with uniform probability on the interval  $[0,1]$ . So, if your program contained the call `import numpy as np`, you can generate a random number between 0 and 1 and assign it to the variable `x` by writing `x = np.random.rand()`. The module `numpy.random` has many more random number generators that you may find useful. See <http://docs.scipy.org/doc/numpy/reference/routines.random.html> for more details.

## 5.5 Scope

The *scope* of a variable, is the region of the program in which it is accesible. Up until now, most of the variables you have created have been accesible everywhere in your program, but variables defined and used within a function are different.

It is possible to have the same variable name for different variables. In terms of the box analogy, it is possible to have boxes with different contents, with the same label as long as Python can determine which should be used in that region of the Program. When Python sees a variable name it applies the “LGB” rule to determine which “box to look in”.

It first looks in the “Local” list of names for the variable. This is the variables defined within the function, e.g. `x` and `y` in the `addnumbers()` example function in Section [Making your own functions](#). However, if the name is not found it then looks in the “Global” list, which is a list of variables accesible from everywhere. If the name is not found there it then resorts to the “Built-in” names (things like `range()` and `input`).

Here is an illustrative example:

```

1 def addnumbers(x,y):
2     spam = "LOCAL SPAM!!"                # Local definition
3     print("spam inside the function is:", spam)
4     return x + y
5
6 spam = "GLOBAL SPAM!!"
7 print(addnumbers(5,10))                  # Function called here
8 print("spam outside the function is:", spam)
```

When the function is called, it does what our original `addnumbers()` function did, i.e. returns the sum of the two parameters. It also prints the contents of the variable `spam`. This is defined locally, so Python does not bother to look in the Global or Built-in lists (where it is defined differently).

When `spam` is printed again *outside* the function `addnumbers()` the first `spam` Python finds is the Global definition. Python cannot descend into the Local list of the function.

This may seem confusing, but it is done for a good reason. The crucial point to take away from this section is one important implication for the way you write your programs:

If you change a variable inside a function, and then inspect it (by printing it for example) outside the function that change will not be reflected:

```
In [31]: def addnumbers(x, y):
.....:     z = 50
.....:     sum = x + y
.....:     return sum

In [32]: z = 3.14
In [33]: print(addnumbers(10, 5))
15
In [34]: print(z)
3.14
```

If you really want these changes to be reflected globally, then tell Python so by making the variable, `z` in this case global.

```
In [35]: def addnumbers(x, y):
.....:     global z
.....:     z = 50
.....:     sum = x + y
.....:     return sum

In [36]: z = 3.14
In [37]: print(addnumbers(10, 5))
15
In [38]: print(z)
50
```

Making variables global can cause subtle problems and should be avoided. As alluded to in the footnote in Section *Making your own functions* you may find it better to pass the result back as a list.

## 5.6 Python differences

### 5.6.1 for loops

As mentioned in Section *for loops*, Python's for loops are actually *foreach* loops: for each element in the list that follows, repeat the nested block that follows the for statement, e.g. :

```
In [39]: for i in [1,2,3,4]:
.....:     print(i)

1
2
3
4
```

This is fundamentally different to the way for loops work in other languages. It allows the index of the for loop (`i` in this example) to step through arbitrary elements.

By using the range function we are replicating the behaviour of for loops in most other languages (C, Pascal, etc.). That is the index is implemented by a constant amount between an lower and upper bound.

However, in Python for loops are much more flexible. Consider the following example:

```
In [40]: for i in [1,2,3,500]:
.....:     print(i)
```

```
1
2
3
500
```

This kind of thing is not easy to do in other languages.

The `for` loop usually steps through a list of items, but it can also be used with arrays:

```
In [41]: from scipy import *
In [42]: xx = array([1,10.2,-509])
In [43]: for i in xx:
.....:     print(i)

1.0
10.2
-509.0
```

This is very useful when writing functions which take array parameters whose size is not known in advance.

## 5.7 Python Modules

### 5.7.1 Importing Python Modules

Writing modules is a convenient way to store and run Python code. As you write more and more code, you are likely to find that you want to reuse certain code in a variety of contexts. Suppose, for example, you want to analyze data from a new set of experiments but you would like to make use of some Python functions you wrote previously for some other experiments. You could copy and paste those old routines that you want to use into a new Python module and thus incorporate them into a new data analysis package appropriate for this experiment. However, Python provides a much more efficient way of incorporating pre-existing Python code into new Python routines. For example, suppose you collect into a single module your own set of useful data analysis routines; let's call this module `crunchthedata.py`. You can gain access to all of the routines contained in `crunchthedata.py` simply by writing, either in a module or at the Python interpreter prompt:

```
import crunchthedata
```

Doing so will allow you access to all the functions defined in the module `crunchthedata.py`. Suppose, for example, that there is a function `messagemydata(x, y, t)` in `crunchthedata.py`. You can call this function from the Python interpreter prompt or from within a module by writing

```
output = crunchthedata.messagemydata(x, y, t)
```

The `crunchthedata.` prefix to `messagemydata(x, y, t)` tells the Python interpreter to look inside the module `crunchthedata` for the function `messagemydata`. You might complain that writing out the `crunchthedata.` prefix everytime you call `messagemydata(x, y, t)` is verbose and annoying. Python provides a couple of alternatives to address this complaint. First, you can create a shorter “nickname” or *alias* for the module by calling it using the syntax:

```
import crunchthedata as ctd
```

Now you call the function by writing

```
output = ctd.messagemydata(x, y, t)
```

Alternative, you can write

```
from crunchthedata import *
```

which imports all the functions defined in `crunchthedata`, or you can write

```
from crunchthedata import massagemydata
```

which imports only the function `massagemydata(x, y, t)`. In either case, no prefix is needed. You simply write

```
output = massagemydata(x, y, t)
```

Note, however, that assigning prefixes is the preferred method (e.g. “`import crunchthedata as ctd`”), and is recommended by most of the Python community. Using prefixes tells the reader where a function comes from. It also avoids function name conflicts. The Python community has developed some set of standard prefixes for various commonly-used packages. Here we provide a few:

```
import scipy as sp
import numpy as np
import matplotlib.pyplot as plt
```

Thus, a call to the `plot` function would be written as `plt.plot(x, y)`. You don’t have to use these conventions, but doing so avoids confusion.

## 5.7.2 Running Python scripts with modules

Suppose the following code below, which we encountered in Section *Scope*, is part of a module named `howmuchspam.py`

```
1 def addnumbers(x,y):
2     spam = "LOCAL SPAM!!"
3     print("spam inside the function is:", spam)
4     return x + y
5
6 spam = "GLOBAL SPAM!!"
7 print(addnumbers(5,10))
8 print("spam outside the function is:", spam)
```

Now suppose we want to use the function `addnumbers(x,y)` defined in the module `howmuchspam.py` from the console. If we type `import howmuchspam`, Python will execute the entire file, including the script below the function definition, which is not what we want to happen. We just want access to the function `addnumbers(x,y)`. Python has a simple mechanism for addressing this problem. We simply rewrite the module `howmuchspam.py` like this:

```
1 def addnumbers(x,y):
2     spam = "LOCAL SPAM!!"
3     print("spam inside the function is:", spam)
4     return x + y
5
6 if __name__=='__main__':           # note pairs of underlines
7     spam = "GLOBAL SPAM!!"
8     print(addnumbers(5,10))
9     print("spam outside the function is:", spam)
```

Now if you write `import howmuchspam`, Python will load the function definition `addnumbers` but will not run the code following the `if __name__=='__main__':` line. That `if` statement directs Python to execute the code following that line only if the `howmuchspam.py` is run as the main routine. That is, typing `run howmuchspam` will cause Python to load the function definition and run the script below the `if __name__=='__main__':`. In general, you should write every module this way if you think there is any chance that you may want to import the

function definitions for use outside a particular module. Finally, take note of the fact that all of the code following the `if __name__ == '__main__':` must be indented, as is the case for any `if` statement.

## 5.8 Python language summary

### 5.8.1 Python functions

Function	Description
<code>print(x)</code>	prints <code>x</code> to terminal
<code>a = input("Type in stuff: ")</code>	Takes input from terminal and sets it equal to variable <code>a</code>
<code>b = raw_input("Type in stuff: ")</code>	Takes input from terminal and sets it equal to string <code>a</code>
<code>range([start,] stop [, step])</code>	Makes a list of integers

### 5.8.2 Arithmetic

Operation	What it does
<code>a + b</code>	addition
<code>a - b</code>	subtraction
<code>a * b</code>	multiplication
<code>a / b</code>	division
<code>a ** b</code>	exponentiation ( <code>a</code> to the power <code>b</code> )
<code>a \%~b</code>	remainder of <code>a/b</code>

### 5.8.3 Comparison tests and Booleans

Comparison	What it tests
<code>a &lt; b</code>	<code>a</code> is less than <code>b</code>
<code>a &lt;= b</code>	<code>a</code> is less than or equal to <code>b</code>
<code>a &gt; b</code>	<code>a</code> is greater than <code>b</code>
<code>a &gt;= b</code>	<code>a</code> is greater than or equal to <code>b</code>
<code>a == b</code>	<code>a</code> is equal to <code>b</code>
<code>a != b</code>	<code>a</code> is not equal to <code>b</code>
<code>a &lt; b &lt; c</code>	<code>a</code> is less than <code>b</code> , which is less than <code>c</code>

### 5.8.4 Python loops & conditionals

```
for x in list:
    print(x)
    ...
```

where `list` is a list or array. First line must end with a colon and subsequent lines in loop must be indented. A `for` loop ends when the indentation ends.

```
1 x = input("Enter a number")
2
3 if 0 <= x <= 10:
4     print("That is between zero and ten inclusive")
```

```

5 elif 10 < x < 20:
6     print("That is between ten and twenty")
7 else:
8     print("That is outside the range zero to twenty")

```

Any valid condition (True/False) statement is permissible in the `if` and `elif` statements. The `elif` and `else` statements are optional.

```

1 i = 1
2 while i < 10:
3     print("i equals:", i)
4     i = i + 1
5
6 print("i is no longer less than ten")

```

A `while` loop keeps executing until the condition in the `while` statement is no longer satisfied or when a `break` statement is encountered. The `break` statement can also be used to end a `for` loop.

## 5.8.5 NumPy functions

See [www.scipy.org/Numpy\\_Example\\_List\\_With\\_Doc](http://www.scipy.org/Numpy_Example_List_With_Doc) for more detailed information and a complete list with examples.

Function	Description
<code>sqrt(x)</code>	Returns the square root of $x$
<code>exp(x)</code>	Return $e^x$
<code>log(x)</code>	Returns the natural log, i.e. $\ln x$
<code>log10(x)</code>	Returns the log to the base 10 of $x$
<code>sin(x)</code>	Returns the sine of $x$
<code>cos(x)</code>	Return the cosine of $x$
<code>tan(x)</code>	Returns the tangent of $x$
<code>arcsin(x)</code>	Return the arc sine of $x$
<code>arccos(x)</code>	Return the arc cosine of $x$
<code>arctan(x)</code>	Return the arc tangent of $x$
<code>fabs(x)</code>	Return the absolute value, i.e. the modulus, of $x$
<code>floor(x)</code>	Rounds a float <i>down</i> to its integer

The following functions create arrays.

Function	Description
<code>array(x)</code>	Creates array from a list $x$
<code>zeros(x)</code>	Creates array of zeros (0) of dimension $x$
<code>ones(x)</code>	Creates array of ones (1) of dimension $x$
<code>rand(N)</code>	Creates array of random numbers on (0, 1) of dimension $N$
<code>eye(N)</code>	Creates $N \times N$ identity matrix
<code>arange([start,] stop [, step])</code>	like range but for (real) arrays
<code>linspace(start, stop, N)</code>	evenly spaced array of $N$ elements from start to stop

## 5.8.6 SciPy FFT functions

These functions for working with discrete Fourier transforms are available from the Python module `scipy.fftpack`.



Function	Description
<code>fft(x)</code>	Returns discrete Fourier transform of $x$
<code>ifft(x)</code>	Returns inverse discrete Fourier transform of $x$
<code>fftshift(x)</code>	Shifts zero-frequency component to the center of the spectrum
<code>ifftshift(x)</code>	Inverse of <code>fftshift</code>

## 5.8.7 Numpy random number generators

The Python module Numpy has a module `numpy.random` that contains a number of useful functions for generating random numbers. We list several below along with a description of what they do. More information can be found at <http://docs.scipy.org/doc/numpy/reference/routines.random.html>. These routines are loaded by writing `from numpy.random import *`. Try them out in the Pylab console before using them in a program to make sure they work as you expect.

Function	Description
<code>normal(mean=0.0, sigma=1.0, size=None)</code>	random floats with Gaussian distribution
<code>random_sample(size=None)</code>	random floats in the half-open interval [0.0, 1.0)
<code>randint(low, high=None, size=None)</code>	random integers on half-open interval [low,high)
<code>lognormal(mean=0.0, sigma=1.0, size=None)</code>	random floats with log-normal distribution
<code>poisson([lamda, size])</code>	random floats with a Poisson distribution

## 5.8.8 Matplotlib plotting functions

These functions are from the Python module `matplotlib.pyplot`. For a basic list of Matplotlib plotting functions with examples, see <http://matplotlib.sourceforge.net/>. You can find a brief but informative tutorial at [http://matplotlib.sourceforge.net/users/pyplot\\_tutorial.html](http://matplotlib.sourceforge.net/users/pyplot_tutorial.html).

Function	Description
<code>figure(N)</code>	Creates new figure pane number $N$
<code>plot(x1, y1, str1, x2, y2, str2)</code>	Creates plot of $(x, y)$ data sets formatted by <code>str</code>
<code>xlabel('strx')</code>	$x$ -axis label
<code>ylabel('stry')</code>	$y$ -axis label
<code>title('stry')</code>	plot title
<code>xlim([a,b])</code>	Sets plotting limits for $x$ axis.
<code>ylim([a,b])</code>	Sets plotting limits for $y$ axis.
<code>subplot(a, b, c)</code>	Sets current subplot to $c$ in figure with $a \times b$ subplots arranged in $a$ rows and $b$ columns
<code>axhline(color = 'k')</code>	Draws horizontal black line at $y = 0$
<code>axvline(color = 'k')</code>	Draws vertical black line at $x = 0$
<code>grid(True)</code>	Draws grid at major axes divisions
<code>legend( ('l1', 'l2', 'l3'), ... loc='upper left')</code>	Draws legend with labels l1, l2, l3 in upper left corner

## 5.9 Taking your interest further

If you would like to know more then the online repository of all things Python-related is <http://www.python.org>. You can get more information about matplotlib at <http://matplotlib.sourceforge.net/>. You may find the tutorial

on `matplotlib` at [http://matplotlib.sourceforge.net/users/pyplot\\_tutorial.html](http://matplotlib.sourceforge.net/users/pyplot_tutorial.html) helpful. You can find a lot of useful Python numerical routines at <http://www.scipy.org/Cookbook>. For those of you interested in getting deeper into Python, I recommend *Learning Python* by Mark Lutz. Either the 3rd or 4th Edition should be fine. The 4th Edition just came out and describes Python 3.0. We are using Python 2.7 in this class because the scientific Python packages have not yet been updated to be compatible with 3.0. The 3rd Edition uses Python 2.5, which for our purposes is indistinguishable from 2.7.

# GETTING HELP WITH PYTHON

## 6.1 Getting help from within IPython

You can get help on a function from within the IPython console by typing `?spam` where `spam` is the name of a function, variable, or module. For example, typing `?range` gives you help on the `range` function:

```
In [142]: ?range
Type:      builtin_function_or_method
Base Class: <type 'builtin_function_or_method'>
String Form: <built-in function range>
Namespace: Python builtin
Docstring:
    range([start,] stop[, step]) -> list of integers

    Return a list containing an arithmetic progression of integers.
    range(i, j) returns [i, i+1, i+2, ..., j-1]; start (!) defaults to 0.
    When step is given, it specifies the increment (or decrement).
    For example, range(4) returns [0, 1, 2, 3]. The end point is omitted!
    These are exactly the valid indices for a list of 4 elements.
```

The help facility tells you what the basic function of the `range` function is and how to use it. Let's try another example:

```
In [140]: ?sin

Base Class:      <type 'numpy.ufunc'>
String Form:     <ufunc 'sin'>
Namespace:      Interactive
File:           /Library/Frameworks/Python.framework/Versions/6.0.0/lib/python2.6/site-packa
Docstring:
    sin(x[, out])

    Trigonometric sine, element-wise.

    Parameters
    -----
    x : array_like
        Angle, in radians (:math:`2 \pi` rad equals 360 degrees).

    Returns
    -----
    y : array_like
        The sine of each element of x.
```

```

See Also
-----
arcsin, sinh, cos

Notes
-----
The sine ...
:

```

The response from the help facility in this case ends with a semicolon `:`. That means that there is more information available, which you can get by pressing the space bar. If you're not interested in getting more information, type `q` at the prompt and the system will return to the IPython console. If you keep pressing the spacebar, you may come to the end of the help text on that copy. If the system doesn't return to the IPython console, press `:` and then press `q`.

ou can also use the help facility to get information about variables and functions you have defined or imported from within the IPython console. You will note that the help facility uses docstrings as part of the its response, which is another good reason for you to use them.

## 6.2 Web Resources for Python

There is no single source on the web for documentation about Python. Instead there are multiple resources, many of them on the web. Here we list some of the more important web sites.

- <http://docs.python.org/>. Official documentation for Python 2.7, the version of Python used in version 7.0 of the Enthought distribution of Python. This documentation is useful sometimes but not the best place to learn Python.
- <http://docs.numpy.org/>. Find links here to the NumPy and SciPy Reference Guides, which direct you to all the official NumPy and SciPy documentation. Links to a few of the more useful NumPy and SciPy libraries are listed below.
- <http://docs.scipy.org/doc/numpy/reference/routines.fft.html>. Fast Fourier Transform routines.
- <http://docs.scipy.org/doc/numpy/reference/routines.random.html>. Random number generators and associated routines.
- <http://www.scipy.org/Cookbook>. Compilation of various Python routines that you might find useful for specialized numerical chores (a bit slow to load). See also [http://www.scipy.org/Topical\\_Software](http://www.scipy.org/Topical_Software).
- <http://matplotlib.sourceforge.net/>. Reference for the matplotlib plotting package. The main page contains a useful list with links to the most common plotting commands. Full documentation is available at <http://matplotlib.sourceforge.net/contents.html>. There is also a useful Pyplot tutorial available at [http://matplotlib.sourceforge.net/users/pyplot\\_tutorial.html](http://matplotlib.sourceforge.net/users/pyplot_tutorial.html), which has a nice list of examples showing how to do some common plotting tasks.
- <http://www.pythonware.com/library/pil/handbook/index.htm>. Documentation of PIL, the Python Imaging Library, a set of Python routines for reading, processing, and writing most common digital image formats (e.g. tiff, jpeg, etc.). SciPy also has a number of useful image processing routines summarized at <http://docs.scipy.org/doc/scipy/reference/ndimage.html>.

## 6.3 Books about Python

There are a wide range of books about Python. The problem is that Python is used for so many different kinds of applications that it can be confusing to determine which is book is best for you. Here are a few you may wish to consider:

- *Learning Python, 3rd Ed.* by Mark Lutz (O'Reilly Media, 2008). This is a very good place to start for those who want a thorough easy-to-understand introduction to Python. There isn't much about scientific applications of Python, but otherwise it is excellent. The 4th edition is ok too but covers Python 3.0. As of March 2011, the SciPy and NumPy packages work with Python 2.7 but not 3.0.
- *Real World Instrumentation with Python* by J. M. Hughes (O'Reilly Media, 2011). This book tells you how to use Python to monitor and control laboratory instrumentation hardware—things like voltmeters, servo motors, *etc.* This is just what you need if you are working in a research laboratory and you want to control your experiment and log data.

# ERRORS

When there is a problem with your code, Python responds with an error message. This is its attempt at explaining the problem. It might look something like this:

```
In [105]: print x
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in ?
    print x
NameError: name 'x' is not defined
```

The first few lines sometimes contain useful information about where Python thinks the error occurred. If you are typing a module (rather than working interactively), click and hold the right mouse button and select `go to file/line`. This will take you to the line Python thinks is the problem. This is not always where the actual problem lies so analyse the last line of the error message too. This Appendix attempts to help you understand these messages.

## 7.1 Attribute Errors, Key Errors, Index Errors

Messages starting “`AttributeError:`”, “`KeyError:`” or “`IndexError:`” generally indicate you were trying to reference or manipulate part of a multi-element variable (lists and arrays) but that element didn’t exist. For example:

```
In [106]: xx = array([1,2])
In [107]: print xx[5]
Traceback (most recent call last):
  File "<pyshell#9>", line 1, in ?
    print xx[5]
IndexError: index out of bounds
```

## 7.2 Name Errors

Name errors indicate that a variable you have referred to does not exist. Check your spelling. You might have mistyped a function, e.g. `printt x`. Check you haven’t attempted to do something with a variable before assigning a value to it, e.g. typing the only the following into a module will not work:

```
print x
x = 5
```

## 7.3 Syntax Errors

These suggest there is a sufficiently severe problem with the way your code is written that Python cannot understand it. Common examples are missing out the colon at the end of a line containing a `for`, `if` or `while` loop; writing a condition with just one `=`, e.g.

```
if x = 5:
    print "x is equal to five"
```

Check that you haven't forgotten to end any strings with quotes and that you have the right number of parantheses. Missing out parentheses can lead to a syntax error on the *next* line. You will get a `SyntaxError` when you run your program if the user does not respond to an `input()` function. Incorrectly indenting your program might also cause `SyntaxErrors`

## 7.4 Type Errors

You have tried to do something to a variable of the wrong type. There are many forms:

**`TypeError: illegal argument type for built-in operation`** You asked Python to do a built-in operation on the wrong type of variable. For example, you tried to add a number to a string.

**`TypeError: not enough arguments; expected 1, got 0`** You used a function without supplying the correct number of parameters.

**`TypeError: unsubscriptable object`** You tried to reference a variable that did not have more than one element (e.g. a float or integer) by offset:

```
In [108]: x = 5
In [109]: print x[0]
```

# RESERVED WORDS

You may not name your variables any of the following words as they mean special things in Python:

and	assert	break	class	continue
def	del	elif	else	except
exec	finally	for	from	global
if	import	in	is	lambda
not	or	pass	print	raise
return	try	while		

Do NOT use any of the following words either (although they are not strictly Python reserved words, they conflict with the names of commonly-used Python functions):

Data	Float	Int	scipy	matplotlib
array	close	float	int	input
open	range	type	write	zeros

You should also avoid all the names defined in the `math` library (you *must* avoid them if you `import` the library):

acos	asin	atan	cos	e
exp	fabs	floor	log	log10
pi	sin	sqrt	tan	



# INSTALLING PYTHON

We will be using the Enthought distribution of Python. This is the easiest way to get going with Python. The distribution is free to academic users. To get the free version, follow the instructions below.

Section *Setting up Python on Windows XP and Windows 7* provides instructions for downloading and installing Python on a PC running Windows XP or Windows 7.

Section *Setting up Python on MacOSX* provides instructions for downloading and installing Python on a Mac running MacOSX (version 10.4 or later).

## 9.1 Setting up Python on Windows XP and Windows 7

### 9.1.1 Downloading and installing Python

Follow the instructions for downloading the Enthought version of Python below. Do not install another version. Do not download from another site! Please follow the directions below.

- Go to <http://www.enthought.com/products/getepd.php>. Scroll down to the bottom of the page and press the light blue button *ACADEMIC*. This takes you to another page.
- Provide your academic e-mail address in the space provided. Enthought will e-mail the web address where you can find the software to download.
- Go to the the web address e-mailed to you by Enthought and download the appropriate file. It is recommended to download the file “epd-7.1-2-win-x86.msi”, which is the 32-bit version, but you may use “epd-7.1-2-win-x86\_64.msi”, the 64-bit version, provided you have the appropriate hardware.

Once you are finished installing Python, you can launch Python from the *start/All Programs/Enthought* menu. You can also launch Python by pressing the **PyLab** icon if you elected to put icons on your desktop and **Quick Launch Toolbar** during installation. You will not need **Mayavi** in this course so you may want to delete these shortcuts from the desktop and **Quick Launch Toolbar**. Go ahead and launch Python by pressing a **PyLab** icon. A Python window should appear with the following prompt:

```
In [1]:
```

At the In [1]: prompt, type 2+3. You should see the following display:

```
In [1]: 2+3
Out[1]: 5
```

```
In [2]:
```

The Python console is ready for your next input.

Since you will be using these applications frequently, you may want to make a shortcut for *PyLab* and place it in *Quick Launch Toolbar* or on the *Desktop*.

### 9.1.2 Setting your default Python text editor to Notepad++

You can use Python “out-of-the-box” as installed above using the Enthought distribution. We will be using the IPython (or PyLab) terminal version of Python as it has the best functionality for scientific computing. Out of the box, however, the default Python editor is **Notepad**, which does not have the functionality required of a good Python editor. A good Python editor to start with is **Notepad++**. It is free, adapted for creating and editing Python programs, and fairly intuitive. The instructions below tell you how to download, install, and set up **Notepad++** so that it works well with Python.

1. Install Enthought Python.
2. **Download and install Notepad++.**
  - Go to <http://sourceforge.net/projects/notepad-plus/files/>. Download the latest version of Notepad++ and install it (as Administrator).
3. **Changing Python’s default editor to Notepad++**
  - Right click on the *My Computer* icon on your desktop and choose *Properties*.
  - For Windows XP: Select the *Advanced* tab.  
For Windows 7: Select *Advanced System Settings* at the upper left hand side of the panel.
  - Press the *Environment Variables* button at the bottom of the panel. This brings up the *Environment Variables* panel.
  - Under *System Variables*, press the *New* button. This brings up the *New System Variable* panel.
  - Under *Variable name*: type `EDITOR`. Be sure to use **ALL CAPITAL LETTERS**.
  - Under *Variable value*: type the path to the **Notepad++** executable file. On most computers this will be `"C:\Program Files\Notepad++\notepad++.exe"`. Because the path name includes spaces, you must include the quotation marks around the path name.
  - Press *OK* to close out all the panels

The net effect of this change should be that when you type `edit`, the in the **PyLab** console, Python invokes **Notepad++** as the default editor. If you want to edit a file named `test.py`, for example, you type `edit test.py`. For this to work, however, the current directory must be set to the same directory as `test.py`. Otherwise you need to specify the path.

4. **Configure Notepad++ for Python**
  - Python uses indentation as an essential part of its syntax, which leads to concise readable code. However, Python can get confused if the indentation is sometimes done by tabbing and other times done by spaces. The simplest way to keep Python happy is to always use spaces to indent. Nevertheless, the tab key is still a nice way to implement consistent indentation. So I suggest that you change the default setting on **Notepad++** so that when you use the tab key, **Notepad++** inserts spaces (4 spaces seems to be the “standard”). Here is how you do it:
  - From the *Settings* menu in **Notepad++**, select *Preferences*
  - In the *Preferences* panel, select the *Language Menu/Tab Settings* tab.
  - In the *Tab Settings* box, scroll down and select *Python* so that it becomes highlighted.
  - If the *Use Default Value* box is ticked, untick it.

- Tick the *Replace by space* box. Leave the *Tab size* at 4.

### 9.1.3 Setting PYTHONPATH

When you try to load or run a Python program (script, module, *etc.*), Python looks in various places for the file. It looks in the current directory, in certain Python system directories, and in certain other directories that you can specify that you can specify by setting a variable called PYTHONPATH.

Before you set PYTHONPATH, create a directory where you want to store all your Python programs. For example, it might be PyProgs located in your My Documents directory. The path for this directory would then be something like C:\Documents and Settings\Cathy\My Documents\PyProgs on Windows XP and ... on Windows 7. We want to set the *environment variable* PYTHONPATH to this path name. To do this, follow these instructions:

- Right click on the *My Computer* icon on your desktop and choose *Properties*.
- For Windows XP: Select the *Advanced* tab.  
For Windows 7: Select *Advanced System Settings*.
- Press the *Environment Variables* button at the bottom of the panel. This brings up the *Environment Variables* panel.
- Under *User Variables*, press the *New* button. This brings up the *New User Variable* panel.
- Under *Variable name*: type PYTHONPATH. Be sure to use **ALL CAPITAL LETTERS**.
- Under *Variable value*: type the path to the name of the directory you want to use for your Python programs. Using the above examples, under Windows XP this might be: "C:\Documents and Settings\Cathy\My Documents\PyProgs". Under Windows 7 this might be: "C:\Users\Cathy\My Documents\PyProgs". If the path name includes spaces (My Documents), you must include the quotation marks around the path name.
- Press *OK* to close out all the panels

## 9.2 Setting up Python on MacOSX

This section provides instructions for downloading Python on machines running MacOSX (*i.e.* version 10.4 of the Mac operating system or later). Please see your instructor if you have an earlier version of the MacOS.

### 9.2.1 Downloading and installing Python

Download the Enthought version of Python. This is the easiest way to get going with Python. The distribution is free to academic users. To get the free version, follow these instructions.

- Go <http://www.enthought.com/products/getepd.php>. Scroll down to the bottom of the page and press the light blue button *ACADEMIC*. This takes you to another page.
- Enter your NYU email address (...@nyu.edu) in the appropriate box (NOT *EPD for PPC* unless you have a very old MAC). A link from which you can download the EPD Python software will be sent to your email address.
- Once you receive the link from EPD, go to the web site and download the version that ends with *macosx-i386.dmg*. This is a big file, about 250 MB, so make sure your computer is connected to a fast internet connection.
- Go to your *Downloads* folder, click on the file *EPD.mpkg* file, and proceed with the installation following the default procedure.

Once you are finished installing Python, go to the Mac *Applications* folder and locate the Enthought Python installation in a folder named *Enthought*. Inside that folder you should find application labeled **PyLab**. Double clicking this icon launches Python. A **Terminal** window should appear with the following prompt:

```
In [1]:
```

If a Terminal window does not appear, double click the **Terminal** application (found in the *Applications/Utilities* folder). Then the Python **Terminal** window should appear with the above prompt. At the `In [1]:` prompt, type `2+3`. You should see the following display:

```
In [1]: 2+3
Out[1]: 5

In [2]:
```

The Python console is ready for your next input.

Since you will be using these applications frequently, you may want to make aliases of the **Terminal** and **PyLab** applications and place them in the Mac **Dock** or on the **Desktop**.

## 9.2.2 Setting your default Python text editor to TextWrangler

You can use Python “out-of-the-box” as installed above using the Enthought distribution. We will be using the IPython terminal version of Python as it has the best functionality for scientific computing. Out of the box, however, the default PyLab editor is **vim**, which is an enhanced version of the **vi** editor. If you already know and like the **vim** editor, skip this part. If you want to use another editor like **emacs**, you can do this too but you should already be an **emacs** expert. The **vim** and **emacs** editors are not at all intuitive so the vast majority of you will be much happier using another text editor. A good choice for the Mac is **TextWrangler**. It is free, adapted for creating and editing Python programs, and fairly intuitive. The instructions below tell you how to download, install, and set up **TextWrangler** so that it works well with Python.

1. Install Enthought Python.
2. **Download and install TextWrangler.**
  - Go to <http://www.barebones.com/products/TextWrangler/download.html> and press the “Download Now” button. Then follow instructions.
  - Find the **TextWrangler** icon (in the Applications folder) and launch the application to make sure it’s working. Do NOT skip this step or you may have problems with the next step!
3. **Modify .bash\_profile (or .profile) file in home directory**
  - Close **PyLab** completely. Then open the Terminal window (*i.e.* the **UNIX** shell, **NOT** the Python console). If you have done this correctly, you should see a `$` prompt. (If you see an `In [1]:` or similar prompt, you have opened the **PyLab** shell rather than the **UNIX** shell. Close the **PyLab** try again.) From the **UNIX** Terminal window, go to your home directory by typing `cd ~` (not including the quotes).
  - Once in your home directory, type `more .bash_profile` (don’t forget the underline character `_` between “bash” and “profile”). You should see something like

```
# Setting PATH for EPD-7.1-2
# The original version is saved in .bash_profile.pysave
PATH="/Library/Frameworks/Python.framework/Versions/Current/bin:${PATH}"
export PATH
```

If you do not see the above text, type `more .profile` You should see something like

```
# Setting PATH for EPD-7.1-2
# The original version is saved in .profile.pysave
PATH="/Library/Frameworks/Python.framework/Versions/Current/bin:${PATH}"
export PATH
```

- Open the file `.profile` or `.bash_profile` by typing “`edit .profile`” or “`edit .bash_profile`” (open the file that contains one of the above texts). This invokes the text editor **TextWrangler**, which you can use to edit the file. Go to the end of the file and add the following line.

```
export EDITOR=edit
```

Be sure to use lower and upper case letters exactly as shown above. Save the file and quit **TextWrangler**.

The net effect of this change should be that when you type `edit`, the in the **PyLab** console, Python invokes **TextWrangler** as the default editor (this should also work from the **Terminal** window). If you want to edit a file named `test.py`, for example, you type `edit test.py`. For this to work, however, the current directory must be set to the same directory as `test.py`. Otherwise you need to specify the path.

#### 4. Configure Textwrangler for Python

- Python uses indentation as an essential part of its syntax, which leads to concise readable code. However, Python can get confused if the indentation is sometimes done by tabbing and other times done by spaces. The simplest way to keep Python happy is to always use spaces to indent. Nevertheless, the tab key is still a nice way to implement consistent indentation. So I suggest that you change the default setting on **TextWrangler** so that when you use the tab key, **TextWrangler** inserts spaces (4 spaces seems to be the “standard”). Here is how you do it: From the Preferences menu in **Textwrangler**, select (highlight) Python under Installed languages. Press the Make Default button and then press the Options... button. In the Options for Python menu, tick *Auto-indent* and *Auto-expand tabs*. Set the *Tab width* to 4 spaces. Save you settings.

### 9.2.3 Removing Python

Once you have installed to latest version of Python, you may want to remove older versions. This isn’t a bad idea as old versions of Python take up considerable space on your disk. Alternatively, you may eventually decide you don’t want Python at all on a given computer. The instructions below illustrate how to remove Enthought Python version 7.0 from your compter. To remove any other version, simply substitute the version number you wish to remove for 7.0 in the commands below.

To remove Enthought Python 7.0 from your Mac, launch the terminal and execute the following **UNIX** commands:

```
$ cd /Library/Frameworks/Python.framework/Versions
$ sudo rm -rf 7.0
```