# Handbook of the Physics Computing Course

Michael Williams

December 3, 2002

# CONTENTS

CHAPTER

**ONE**

# Introduction

## 1.1 The Python trial

Thank you for participating in this trial of a proposed new first year computing course. During this trial we would like you to consult demonstrators as much as you like. It is being run for us to iron out the problems in the course, and so we will be most appreciative of any feedback that you can give us about the course or the handbook. Of course, we also hope you will find the trial interesting and useful!

Like the Pascal course you did last year this trial consists of the following elements:

**Using the system** Logging in; using the graphical interface; files; using the Python interpreter; creating, editing, storing and running programs. You will learn some of the skills associated with the use of the UNIX operating system.

**The elements of Python** Learning enough of a programming language to write simple but useful programs, while being introduced to the concepts of procedural programming.

To introduce yourself to the basics of Python you will need to read through Chapter 3, "The elements of Python" and do the exercises given as you go along. If you do not understand how to tackle a particular exercise ***please* consult a demonstrator**.

The handbook should contain all the information you need to complete the exercises and the problem you will be doing.

**Programming Problems** Once you have gone through the introduction to Python we would like you to attempt a more substantial problem. You may select one from the following list (you may select the one you did in the first year if you wish):

**CO11** Quadratic Equation; straight line fitting by least squares

**CO12** Nuclear decay and the Doppler effect

**CO13** Numerical Integration by Simpson's rule and by a Monte Carlo method

**CO14** Solution of non-linear equations; solution of a differential equation

**CO15** Polynomial curve fitting by least squares

**CO16** Graph plotting; Fourier Analysis

**Note:** whichever problem you choose, for the purposes of the trial it will be recorded as experiment CO91

Whilst it is not required, we would like you to use your logbooks to make notes as you do the course; in any case, you will not need to produce a write-up of the problem you attempt. Rather, we will ask you to discuss it with a demonstrator and fill in a questionnaire.

This course deliberately covers only a subset of Python, which is an extremely powerful and flexible language. In doing the course you should acquire skills applicable both to Python and to programming in general. Python has

been chosen because it is possible to teach enough of the language to write useful programs relatively quickly, yet it is potentially a very powerful language, which is being increasingly used throughout academia and the commercial software industry.

## 1.2 Typographical conventions

This handbook uses the following typographical conventions:

Most of the document appears in the Times font.

`Typewriter font` is used for anything typed, e.g. Python code or UNIX commands.

Sans serif is used for menu and application names

New terms and other pieces of jargon are written in *italics* the first time they are used. [These will be described in a glossary in a later edition].

CHAPTER

**TWO**

# Using the system

## 2.1 Logging in

Sit at one of the Sun terminals. Move the mouse: the screen should come on; if it doesn't after a few seconds, press the power button in the bottom-right corner of the monitor, so the green LED is lit.

You should see a login panel: using lower case, enter your usual Physics username where indicated and then press `Return` **but *before* you enter your password**, ensure that the graphic on the right says "Solaris Common Desktop Environment". If it does not, then click and hold the Options button, and select Common Desktop Environment from the Session menu. Now enter your password and press `Return`.

The login panel should disappear and after a few seconds, you should see various windows appear on the screen. If you want to be able to use the Netscape web browser—which we recommend—click Accept in its License panel (a Netscape browser window will then appear, which you can minimise by clicking on the "dot" button at the top-right of its window). You should also mimimise the Help Viewer and File Manager windows in the same manner. You should now be left with a single Terminal window.

If you do not see a Terminal window, then you can get a new one at any time by clicking once on the terminal icon at the bottom-right of the screen (it should be the third icon from the right). Click in the Terminal window so its border turns dark pink (this indicates that you can type things into it). Terminals provide access to what is known as the UNIX command-line, or *shell*. A Unix shell is a bit like the DOS-prompt in Microsoft Windows, but it is far more powerful and flexible: it makes available hundreds of commands, utilities and programs. The shell indicates that it is ready to receive commands by displaying the following [1] prompt:

```
rayleigh%
```

Some basic help and information about Unix commands is available by typing `help` in a Terminal window. Mostly, however, you will not need to use the shell because we have made available a single, integrated application for doing Python programming called IDLE.

## 2.2 IDLE basics

This section describes the IDLE application that you will be using when you start writing your own Python programs in the next chapter. You will probably need to refer back to it when you start working through Chapter 3.

IDLE is Python's Integrated DeveLopment Environment; in other words it is an application which enables you to both write and run Python programs. [2]

To start up IDLE type **`idle`** at the shell prompt and press the `Return` key:

[1] This is the default shell prompt: it can be customized.

[2] If you are familiar with UNIX or Linux you may alternatively use your favourite UNIX text-editor to write your Python programs, and then run them yourself using the `python` command. However, IDLE does provide a nice environment for developing Python code, so we recommend you try using it.

```
rayleigh% idle
```

**Note:** `idle` is typed entirely in lower case—see Section 2.4, “Case sensitivity”.

At this point, you may wish to **minimise** the Terminal window; **do NOT close it** or IDLE will be killed!

After a few seconds a new window called Python Shell will appear. This is IDLE’s interface to the Python *interpreter*. An interpreter is a program which translates the program *source code* you enter into *machine code* which the computer can understand and run. Chapter 3 explains all about using the Python interpreter.

IDLE also includes an *editor* which is an application, rather like a simple word-processor, that you use to write programs (or indeed any kind of text). IDLE’s editor has been customized to make it particularly useful for writing Python programs, and it should be fairly intuitive to use. Documents are opened, closed and saved by selecting commands in the File menu.

### 2.2.1 Creating a new program

Selecting New Window from the File menu will open an empty editor window called Untitled.

### 2.2.2 Opening existing programs

You can retrieve a previously saved program by choosing Open from the File menu and selecting the file to open.

### 2.2.3 Saving programs

After typing or editing a Python program, save the file by selecting Save from the File menu. The first time you save a file, Save will actually behave like Save As; in other words, a small window will appear where you can enter the name you wish to give your program. On subsequent occasions, Save will simply save the file. Once saved, the program remains on-screen for further editing if you wish.

### 2.2.4 Closing programs

If you really have finished editing your program then choose Close from the File menu. If there are unsaved changes you will be prompted to save them.

In general, when you have finished with a window you should close it rather than minimising it as otherwise your screen will become cluttered with lots of minimised window icons.

## 2.3 Programming using IDLE

As you will see in Section 3.2 it is possible to write programs either interactively or by using the editor. You will probably need to refer back to this section when you start writing your own programs.

### 2.3.1 Using the interactive interpreter

Many of the examples in this handbook use Python’s *interactive interpreter*. If you don’t already have one open, you can get an interactive window (a *Python Shell*) by selecting Python Shell from the Run menu.

A new window containing the following text will appear:

```
Python 2.2.1 (#1, May  2 2002, 23:54:03)
[GCC 2.95.2 19991024 (release)] on sunos5
Type "copyright", "credits" or "license" for more information.
IDLE Fork 0.8 -- press F1 for help
>>>
```

The first four lines are version information and can safely be ignored. `>>>` is the Python prompt. It is here that you type Python code when using it interactively.

The interpreter will highlight your program to aid you: key words it recognizes will be coloured orange while words it does not will be black; quoted text will appear in green and errors in red. This may help you to spot mistakes in your program.

The interpreter will also try to help you write Python in other ways, in particular with laying out your code. As described in Section 3.6.1, code layout is particularly important in Python.

It also has simple *history* capabilities, allowing you to recall recently typed commands and edit and re-use them. To recall a command, use the arrow keys to move the cursor back up to the line you would like to repeat and press the `Return` key. That line will then appear at your prompt. You can now edit it if required and execute it as normal.

### 2.3.2 Using the editor

The editor is used to create and save new programs or to open, modify and save existing stored programs (in Python, stored programs are usually called *modules*). If you need a new editor window then choose New Window from the File menu.

To run a program select Run Program from the Run menu (or press F5). The results of the program will appear in a special Output window.

The editor is customised for writing Python programs: it will highlight and lay out your code for you in the same way as the interpreter.

## 2.4 Case sensitivity

If an operating system or programming language differentiates between a word typed partly in upper case, one typed entirely in upper case, and one typed entirely in lower case it is said to be *case-sensitive*.

You may have used Microsoft Windows: in general it is not case-sensitive. This system runs Solaris which is a variant of the UNIX operating system. UNIX operating systems are always case-sensitive. For example, look what happens if we capitalise the `idle` command:

```
rayleigh% Idle
Idle: Command not found
rayleigh% IDLE
IDLE: Command not found
```

UNIX commands tend to be written entirely in lower case (though sometimes upper-case characters are used as well). Most modern programming languages (eg. C, C++, Java, Python, Perl, etc.) are case-sensitive too. When writing programs in these languages you have to use lower case much of the time.

In Python there are a few exceptions to this rule. The most important examples you will encounter in this course are `Numeric`, `Oxphys`, `Gnuplot`, `Int` and `Float`.

`Numeric` and `Gnuplot` are capitalised because they are the names of other programs. `Int` and `Float` are capitalised to distinguish them from the terms `int` and `float` which have different meanings in Python. The basic rule is that you should type commands in exactly as they appear in this manual.

CHAPTER

**THREE**

# The elements of Python

These notes give a basic introduction to programming using Python and should contain all the information you need for the first-year course.

## 3.1 Hello world

Programming languages are traditionally introduced with a trivial example that does nothing more than write the words *Hello world* on the screen. The intention is to illustrate the essential components of the language, and familiarise the user with the details of entering and running programs.

The good news is that in Python this is an extremely simple program:

```
print "Hello world"
```

The program is self-explanatory: note that we indicated what we wanted to print to the screen by enclosing it in quotation marks.

**EXERCISE 3.1**
**Start up Python's interactive *interpreter* (see Chapter 2, "Using the system", for more details). Get Python to say "Hello world" to you by typing the above command, followed by the `Return` key (this tells Python you have finished that instruction). If you have trouble getting a Python prompt, or get an error message when your program is run, ask a demonstrator.**

The disadvantage of the Python "Hello world" program being so simple is that we haven't learnt much about the language! The next section contains a more instructive example.

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

There are two ways of using Python: either using its *interactive interpreter* as you have just done, or by writing *modules*. The interpreter is useful for small snippets of programs we want to try out. In general, all the examples you see in this book can be typed at the interactive prompt (>>>). You should get into the habit of trying things out at the prompt: you can do no harm, and it is a good way of experimenting.

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 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.

When you start up IDLE, you should see the Python interactive interpreter. You can always recognise an interpreter window by the >>> prompt whereas new module windows are empty. IDLE only ever creates one interpreter window:

if you close it and need to get the interpreter back, select Python shell from the Run menu. You can have multiple module windows open simultaneously: as described in Chapter 2 each one is really an *editor* which allows you to enter and modify your program code (because of this, they will often be referred to in this handbook as *editor windows*). To get a new empty module (editor) window select New window in the File menu.

Here is an example of a complete Python module. Type it into an editor window and run it by choosing Run from the Run menu (or press the F5 key on your keyboard)[1].

```
print "Please give a number: "
a = input()
print "And another: "
b = input()
print "The sum of these numbers is: "
print a + b
```

If you get errors then check through your copy for small mistakes like missing punctuation marks. Having run the program it should be apparent how it works. The only thing which might not be obvious are the lines with `input()`. `input()` is a *function* which allows a user to type in a number and returns what they enter 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 recognised as a Python module, e.g. `prog.py`. Furthermore, **do NOT use spaces in filenames or directory (folder) names**.

**EXERCISE 3.2**
**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.**

## 3.3 Variables

### 3.3.1 Names and Assignment

In Section 3.2 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`. Where relevant you should give your variables a descriptive name, such as `firstname` or `height` [2]. 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 B.

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:

```
>>> x = 5
>>> print x
5
```

[1] You might write programs that seem to stop responding. If this happens try selecting Stop Program from the Run menu

[2] 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. As this is a document written by deadly serious scientists we will avoid this convention (mostly). However, it's something to be aware of if you take your interest in Python further.

You assign 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* [3] 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`" (or, if you prefer, as "`x` *becomes* `5`"). However, we can still do many of things you might do in algebra, like:

```
>>> a = b = c = 0
```

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

```
>>> print a, b, c
0 0 0
```

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

```
>>> i = 2
>>> i = i + 1
>>> print i
3
```

The second line in this example is not possible in maths, 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.

### 3.3.2 Types

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

**Integer**: Any whole number:

```
>>> myinteger = 0
>>> myinteger = 15
>>> myinteger = -23
>>> myinteger = 2378
```

**Float**: A floating point number, i.e. a non-integer.

```
>>> myfloat = 0.1
>>> myfloat = 2.0
>>> myfloat = 3.14159256
>>> myfloat = 1.6e-19
>>> myfloat = 3e8
```

[3] The *syntax* of a language refers to its grammar and structure: ie. the order and way in which things are written.

Note that although 2 is an integer, by writing it as `2.0` we indicate that we want it stored as a float, with the precision that entails.[4] The last examples use exponentials, and in maths 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.

**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.

```
>>> mystring = "Here is a string"
>>> mystring = 'Here is another'
```

**Arrays and Lists**: These are types which contain more than one element, analogous to vectors and matrices in mathematics. Their discussion is deferred until Section 3.10 "Arrays". For the time being, it is sufficient to know that a list is written by enclosing it in square brackets as follows: `mylist = [1, 2, 3, 5]`

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

```
>>> type(mystring)
<type 'str'>
```

`'str'` tells you it is a string. You might also get `<type 'int'>` (integer) and `<type 'float'>` (float) [5].

**EXERCISE 3.3**
**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`:**

```
>>> a = 1.2
>>> print A
```

As a beginner you will avoid making mistakes if you restrict yourself to using lower-case for your Python functions and the names of your variables (but see also Section 2.4 on case-sensitivity).

## 3.4 Input and output

Computer programs generally involve interaction with the user. The is 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 3.2. Functions will be discussed in more detail in Section 3.9, 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 example in Section 3.2 could be rewritten as follows:

```
a = input("Please give a number: ")
```

[4] floats can actually be less *accurate* than ints, however they can store fractional values and hence store numbers to greater *precision*; ints store exact values but are limited to whole numbers. eg. "g=10" is less precise than "g=9.30665" but it is a much more accurate value of the acceleration due to gravity at the Earth's surface!

[5] 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 catch you out if you've programmed before. See Section 4.5, "Python differences".

```
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` command can print several things, which we separate with a comma, as above. If the `print` command 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):

```
>>> x = "Spanish Inquisition"
>>> 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:

```
>>> name = raw_input("Please tell me your name: ")
```

**EXERCISE 3.4**
**Copy the example in Section 3.2 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, personalise the output message; eg. 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

The programs you write will nearly always use numbers. Python can manipulate numbers in much the same way as a calculator (as well as in the much more complex and powerful ways you'll use later). Basic arithmetic calculations are expressed using Python's (mostly obvious) *arithmetic operators*.

```
>>> a = 2               # Set up some variables to play with
>>> b = 5
>>> print a + b
7
>>> print a - b         # Negative numbers are displayed as expected
-3
>>> print a * b         # Multiplication is done with a *
10
>>> print a / b         # Division is with a forward slash /
0.4
>>> print a ** b        # The power operation is done with **
32
>>> print b % a         # The % operator finds the remainder of a division
1
>>> print 4.5 % 2       # The % operator works with floats 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 handbook 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. Well written comments will save you a lot of time. Furthermore, the demonstrators will be able to understand your program more quickly (and therefore mark you more quickly too!).

The rules of precedence are much the same as with calculators. ie. Python generally evaluates expressions from left to right, but things enclosed in brackets are calculated first, followed by multiplications and divisions, followed by additions and subtractions. If in doubt add some parentheses:

```
>>> print 2 + 3 * 4
14
>>> print 2 + (3 * 4)
14
>>> print (2 + 3) * 4
20
```

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

```
>>> 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. Ask a demonstrator if you need more help with this.**

**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 $\pi$. 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 (where the number is known at the start of the loop). Here is an example:

```
sumsquares = 0    # sumsquares must have a value because we increment
                  # it later.

for i in [0, 1, 2, 3, 4, 5]:
    print "i now equal to:", i
    sumsquares = sumsquares + i**2    # sumsquares incremented here
    print "sum of squares now equal to:", sumsquares
    print "------"
```

```
print "Done."
```

The indentation of this program is essential. Copy it into an empty module. IDLE will try and help you with the indentation but if it doesn't get it right use the TAB key[6]. Don't forget the colon at the end of the `for` line. The role of indentation is explained below but first let us consider the `for` loop. Try and work out what a for loop does from the program's output.

The `for` loop can actually be considered as a *foreach* loop: "For each thing in the list that follows, execute some 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`. That is to say, 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. When you do this, the condition will be tested and if appropriate the nested block 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 should give `range()` at least two[7] *parameters*[8]: first the number at which the list starts and then the number up to which the list should go. **Note that the second 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 (3.6.1) using `range(6)` since:

```
>>> print range(0, 6)
[0, 1, 2, 3, 4, 5]
```

So we could have constructed the `for` loop as follows:

```
for i in range(0, 6):
    print "i now equal to:", i
    sumsquares = sumsquares + i**2
    # etc.
```

Here is another example of the use of `range()`:

```
>>> print range(5, 10)
[5, 6, 7, 8, 9]
```

[6] You can also use any number of spaces, as long as the number is consistent.
[7] `range()` can actually be used with just one parameter (the number to stop at), but you will make less mistakes if you use two.
[8] See Section 3.9 for a discussion of parameters: for now it is enough to know that they are the things placed within the parentheses.

If you are uncertain what parametes 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.

```
>>> print range(10, 20, 2)
[10, 12, 14, 16, 18]
```

Note that this goes up to 18, not 19, as the step is 2.

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 1, separated by 0.1 some ingenuity is required:

```
for i in range(0, 10):
    print i / 10
```

**EXERCISE 3.6**
**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 $1/x$ from 1 up to the number the user typed, in steps of, say, 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 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 maths 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 it 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 maths. The possibilites are shown below:

| Comparison | What it tests |
|---|---|
| `a < b` | `a` is less than `b` |
| `a <= b` | `a` is less than or equal to `b` |
| `a > b` | `a` is greater than `b` |
| `a >= b` | `a` is greater than or equal to `b` |
| `a == b` | `a` is equal to `b` |
| `a != b` | `a` is not equal to `b` |
| `a < b < c` | `a` is less than `b`, which is less than `c` |

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`:

```
if x == 10 and y > z:
    print "Some statements which only get executed if"
    print "x is equal to 10 AND y is greater than z."
if x == 10 or y > z:
    print "Some statements which get executed if either
    print "x is equal to 10 OR y is greater than z"
```

These comparisons can apply to strings too[9]. The most common way you might use string comparions is to ask a user a yes/no question[10]:

```
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.**

### 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

[9] although something like `mystring > "hello"` is not generally a meaningful thing to do.
[10] Note string comparisons are case sensitive so `"y" != "Y"`.

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:

```
x = input("Enter a number")

if 0 <= x <= 10:
    print "That is between zero and ten inclusive"
elif 10 < x < 20:
    print "That is between ten and twenty"
else:
    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.**

**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 whilst 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:

```
i = 1
while i < 10:
    print "i equals:", i
    i = i + 1
```

```
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 `[conditon]` used at the top of the while loop is of the same form as those used in `if` statements: see Section 3.7.2, “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:

```
print """
This program returns the smallest non-unity
factor (except one!) of a number entered by the user

"""
n = input("number: ")
i = 2                       # Start at two -- one is a factor of
                            # everything

while (n % i) != 0:         # i.e. as long as the remainder of n / i
   i = i + 1                # is non-zero, keep incrementing i by 1.

# once control has passed to the rest of the program we know i is a
# factor.

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 big primes 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 which can be used for common programming tasks. You can also create your own (see Section 3.11, “Making Functions”). A function is just some Python code which is seperated from the rest of the program. This has several advantages: Repeated sections of code can be re-used without rewriting them many times, making your program clearer. Furthermore, if a function is separated from the rest of the program it provides a conceptual separation for the person writing it, so they can concentrate on either the function, or the rest of the program.

Python has some *built-in* functions, for example `type()` and `range()` that we have already used. These 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 maths, 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. Refer to the discussion of the individual functions for details.

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

However, the functions in the library are contained in separate *modules*, similar to the ones you have been writing and saving in the editor so 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 module for us is the `math` library[11]. If you want to use the functions it contains, put the line `from math import *` at the top of your program.

The `math` functions are then accesible in the same way as the built in functions. For example, to calculate the sin and cos of $\pi/3$ we would write a module like this:

```
from math import *
mynumber = pi / 3

print sin(mynumber)
print cos(mynumber)
```

The `math` module contains many functions, the most useful of which are listed below. Remember that to use them you must `from math import *`.

| Function | Description |
|---|---|
| `sqrt(`$x$`)` | Returns the square root of $x$ |
| `exp(`$x$`)` | Return $e^x$ |
| `log(`$x$`)` | Returns the natural log, i.e. $\ln x$ |
| `log10(`$x$`)` | Returns the log to the base 10 of $x$ |
| `sin(`$x$`)` | Returns the sine of $x$ |
| `cos(`$x$`)` | Return the cosine of $x$ |
| `tan(`$x$`)` | Returns the tangent of $x$ |
| `asin(`$x$`)` | Return the arc sine of $x$ |
| `acos(`$x$`)` | Return the arc cosine of $x$ |
| `atan(`$x$`)` | Return the arc tangent of $x$ |
| `fabs(`$x$`)` | Return the absolute value, i.e. the modulus, of $x$ |
| `floor(`$x$`)` | Rounds a float *down* to its integer |

The `math` 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

```
>>> floor(-3.01)
-4.0
```

**EXERCISE 3.9**
**Use the `math` 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.**

[11] Sorry about the spelling of "`math`". Modern computer languages are generally written according to US English spelling conventions. You do the math!

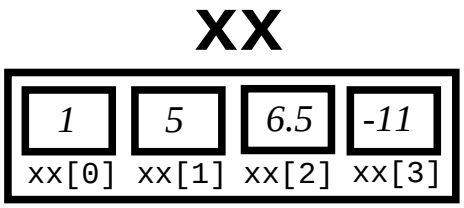


Figure 3.1: Arrays can be thought of as boxes around boxes

## 3.10 Arrays

The elements of a list can, in principle, be of different types, e.g. `[1, 3.5, "boo!"]`. This is sometimes useful but, as scientists you will mostly deal with *arrays*. These are like lists but each element is of the same type (either integers or floats). This speeds up their mathematical manipulation by several orders of magnitude.

Arrays are not a "core" data type like integers, floating points and strings. In order to have access to the `array` type we must *import* the Numeric library. This is done by adding the following line to the start of every program in which arrays are used:

```
from Numeric import *
```

When you create an array you must then explicitly tell Python you are doing so as follows:

```
>>> from Numeric import *
>>> xx = array([1, 5, 6.5, -11])
>>> 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.

The decimal point at the end of 1, 5 and -11 when they are printed indicates they are now being stored as floating point values; all the elements of an array must be of the same type and we have included 6.5 in the array so Python automatically used floats.

We can extend the box analogy used to describe variables in Section 3.3 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.

This simplifies the program—there need not be very many differently named variables. More importantly it 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]` (n.b. *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:

```
>>> print xx
[  1.    5.    6.5 -11. ]
>>> print xx[0]
1.0
>>> print xx[3]
-11.0
>>> print range(xx[1])       # Using the element just like any other
[0, 1, 2, 3, 4]              # variable
```

```
>>> xx[0] = 66.7
>>> 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-Muller tube during succesive one minute intervals. The following program will read those numbers in from the keyboard. and store them in an array.

```
from Numeric import *

counts = zeros(5, Int)      # See below for an explanation of this

for i in range(0, 5):
    print "Minute number", i
    response = input("Give the number of counts made in the minute")
    counts[i] = response

print "Thank you"
```

The contents of the `for` loop are executed five times (see Section 3.6.2 "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:

```
>>> xx = array([3, 4])
>>> 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-Muller 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). This can be `Int` or `Float` (Note the upper case "`I`" and "`F`" — this is to distinguish them from the `float()` and `int()` functions discussed in Section 3.12 "File input and output").

In the Geiger-Muller 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:

```
>>> xx = zeros(5, Int)
>>> print xx
[0 0 0 0 0]
>>> yy = zeros(4, Float)
>>> print yy
[ 0.  0.  0.  0.]
```

If there is any uncertainty as to whether `Int` or `Float` arrays are appropriate then use `Float`.

**EXERCISE 3.10**
**Using `for` loops, `range()`, and the `zeros()` function, construct two 100 element arrays, such that element `i` of one array contains $\sin(2\pi i/100))$ and the corresponding element of the other contains $\cos((2\pi i/100))$.**

**Compute the scalar (i.e. dot) products of the two arrays, to check that $\sin$ and $\cos$ are orthogonal, i.e. their dot product is zero. The scalar, or dot, product is defined as:**

$$\mathbf{x}.\mathbf{y} = \sum_i x_i.y_i$$

**Note:** We have only considered the features of arrays that are common to most other programming languages. However, Python's arrays are extremely powerful and can do some stuff that would have to be done "manually" (perhaps using `for` loops) in other languages. If you find you are using arrays in the problem then it is worth taking a look at Section 4.2, "Arrays". You will also note there is a function in the `Numeric` library that will calculate the dot product of two arrays for you! We want you to do it the hard way though.

## 3.11 Making your own functions

As we saw in Section 3.9, functions are very useful tools for making your programs more concise and modular.

The libraries provide a useful range of facilities but a programmer will often want or need to write their own functions if, for example, one particular section of a program is to be used several times, or if a section forms a logically complete unit.

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 it is *passed*:

```
>>> def addnumbers(x, y):
        sum = x + y
        return sum

>>> x = addnumbers(5, 10)
>>> print x
15
```

The structure of the definition is as follows:

1. 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. [12].
2. Then an indented code block follows. This is what is executed when the function is *called*, i.e. used.
3. 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:

[12]These are known as *formal parameters* (`x` and `y` in this case). The *actual parameters* (`5` and `10` in the example) are assigned to the formal parameters when the function is called. Parameters are also often referred to as *arguments*

```
>>> def addnumbers(x, y):
        sum = x + y
        x = 1000
        return sum

>>> x = 5
>>> y = 10
>>> answer = addnumbers(x, y)
>>> print x, y, answer
5 10 15
```

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 variables. This is done to minimise the risk of subtle errors in your program

If you really want a change to be reflected then return a list of the new values as the result of your function. Lists can then be accessed by offset in the same way as arrays:

```
>>> def addnumbers(x, y):
        sum = x + y
        x = 100000
        return [sum, x]

>>> x = 5
>>> y = 10
>>> answer = addnumbers(x, y)
>>> print answer[0]
15
>>> print answer[1]
100000
```

A fuller discussion of the relationship between variables used in functions and in the main program is discussed in Section 4.4, “Scope”.

## 3.12 File input and output

So far we have taken input from the keyboard and given output to the screen. However, You may want to save the results of a calculation for later use or read in data from a file for Python to manipulate. You give Python access to a file by `open`ing it:

```
>>> fout = open("results.dat", "w")
```

`fout` is then a variable like the integers, floats and arrays we have been using so far—`fout` is a conventional name for an output file variable, but you are free to choose something more descriptive. The `open` function takes two parameters. 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 |

There are various ways of reading data in from a file. For example, the `readline()` *method* returns the first line the first time it is called, and then the second line the second time it is called, and so on, until the end of the file is reached when it returns an empty string:

```
>>> fin = open("input.dat", "r")
>>> fin.readline()
'10\n'
>>> fin.readline()
'20\n'
>>> fin.readline()
''
```

(The `\n` characters are newline characters.)

Note the parentheses are required. If they are not included the file will not be read. They are to tell Python that you are using the `readline()` function—a function is always followed by parentheses, whether it takes any arguments or not.

You can see from the example that you tell Python to use methods by adding a full stop followed by the name of the method to the variable name of the file. This syntax may seem strange[13] but for now just use the examples below as your guide to the syntax, and don't worry about what it means.

The contents are read in as a string but if the data is numeric you need to *coerce* them into either floats or integers before they are used in calculations:

```
>>> fin = open("input.dat", "r")
>>> x = fin.readline()
>>> type(x)
<type 'str'>
>>> y = float(x)
>>> type(y)
<type 'float'>
>>> print y
10.0
```

You can also use the `readlines()` method (note the plural) to read in *all* the lines in a file in one go, returning a list:

```
>>> fin.readlines()
['This is the first line of the file.\n', 'Second line of the file\n']
```

To output or write to a file use the `write()` method. It takes one parameter—the string to be written. If you want to start a new line after writing the data, add a `\n` character to the end:

```
>>> fout = open("output.dat", "w")
>>> fout.write("This is a test\n")
>>> fout.write("And here is another line\n")
>>> fout.close()
```

Note that in order to commit changes to a file, you must `close()` files as above.

[13] It is actually the first you have really seen of *Object Orientated Programming*, a powerful way of programming and thinking about problems that will not be discussed in this course.

`write()` must be given a string to write. Attempts to write integers, floats or arrays will fail:

```
>>> fout = open("output.dat", "w")
>>> fout.write(10)
Traceback (most recent call last):
  File "<pyshell#65>", line 1, in ?
    fout.write(10)
TypeError: argument 1 must be string or read-only character
buffer, not int
```

You must coerce numbers into strings using the `str()` function:

```
>>> x = 4.1
>>> print x
4.1
>>> str(x)
'4.1'
```

If you are trying to produce a table then you may find the `\t` character useful. It inserts a *tab* character, which will move the cursor forward to the next tabstop[14]. It is an example of a non-printed character. It leaves only *white space*—no characters are printed as such. However, it is still a string character, and must be enclosed in quotes. For example, to print the variables `a` and `b` separated by a tab you would type:

```
print a + "\t" + b
```

This is not a perfect method of producing a table. If you are interested in the "right way" of doing this, then ask a demonstrator about *formatted output*.

**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 the first element of both arrays, the second line the second element, an so on. Keep the file as we will use it later.**

## 3.13 Putting it all together

This section shows a complete, well commented program to indicate how most of the ideas discussed so far (Variables, Arrays, Files, etc.) are used together.

Below is a rewritten version of the example in Section 3.2, 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.

```
from Numeric import *

def addnumbers(x, y):   # Declare functions first.
    sum = x + y
    return sum

def getresponse():
```

[14]Tabstops are generally separated by eight characters.

```
    # Create a two element array to store the numbers
    # the user gives. The array is one of floating
    # point numbers because we do not know in advance
    # whether the user will want to add integers or
    # floating point numbers.
    response = zeros(2, Float)
    # Put the first number in the first element of
    # the list:
    response[0] = input("Please give a number: ")
    # Put the second number in the second element:
    response[1] = input("And another: ")
    # And return the array to the rest of the program
    return response

# Allow the user to name the file. Remember this is a string
# and not a number so raw_input is used.
filename = raw_input("What file would you like to store the result in?")

# Set up the file for writing:
output = open(filename, "w")

# Put the users response (which is what the getresponse() function
# returns into a variable called numbers
numbers = getresponse()

# Add the two elements of the array together using the addnumbers()
# function
answer = addnumbers(numbers[0], numbers[1])

# Turn the answer into a string and write it to file
stringanswer = str(answer)
output.write(stringanswer)

# And finally, don't forget to close the file!
output.close()
```

**EXERCISE 3.13**
**The following function computes $e^x$ by summing the Taylor series expansion to $n$ terms. Write a program to print a table of $e^x$ using both this function and the `exp()` function from the `math` library, for x = 0 to 1 in steps of 0.1. The program should ask the user what value of $n$ to use.**

```
def taylor(x, n):
    sum = 1
    term = 1
    for i in range(1, n):
        term = term * x / i
        sum = sum + term
    return sum
```

This is the end of the introduction to Python. Talk to a demonstrator who will be able to suggest a problem for you to attempt (or you may choose the one you did last year but please talk to a demonstartor first). If the problem requires graphical output then you will need to refer to Section 4.1 which provides an introduction to the use of the Gnuplot package from within Python. The rest of Chapter 4 discusses material that is either considered to be "additional" to the basic core knowledge presented in this chapter or that is peculiar to programming in Python. Whilst it is *not* required reading at this stage, you should at least glance over it now so that you can refer to it later if necessary.

CHAPTER

**FOUR**

# Graphical output and additional Python

## 4.1 Graphical output

Generating graphs is not part of core Python so it was not included in the main chapter of this handbook. In this course you will use the Gnuplot package to generate graphical output. It is an extremely powerful and flexible program, although it does have a rather steep learning curve for beginners. However, you will use the Gnuplot.py package to allow you to control Gnuplot from within Python which makes things much easier.

In any program from which you would like graphical output you must include the line...

```
from Oxphys import *
```

...in the same way as you import the `math` and `Numeric` libraries to have access to mathematical functions and arrays respectively. The `Oxphys` library has been written especially for this course. You will find it gives you access to the `math`, `Numeric` and `Gnuplot` libraries, and a special function to print your graphs.

And then, in the same way as you use the `open()` function to get a file ready, use the `Gnuplot()` function to prepare a graph variable:

```
g = Gnuplot()
```

`g` is merely a suggested name for your graph—you are free to call it anything you would call any other variable.

Your program should then prepare two arrays to be plotted: one for the $x$-cooridinates, one for the $y$. Let us assume two arrays exist. `xx` contains the $x$-coordinates of the data, and `yy` the y coordinates.

You then create a variable that points to the data using the `Data() function`, which takes two array paramters. This creates a compound variable type that holds all the information required to generate a plot. Here we call our data variable `results`:

```
i
    results = Data(xx, yy)
```

You can then draw the plot to the screen using the command:

```
g.plot(results)
```

The `plot` method takes one parameter: the `Data` variable we set up in the previous example.

This generates a simple, unlabelled plot. In general you will want to give your graphs titles and label the axes. Returning to the Geiger-Muller tube example, we might label our graph using the following commands:

```
g.title("Geiger-Muller counts made in one minute")
g.xlabel("Time (minutes)")
g.ylabel("Counts")
```

If you would like to join the points on your graph with straight lines then, assuming your graph variable is called `g`, use the following command:

```
g("set data style lines points")
```

This must be done *before* the graph is plotted using the `g.plot` command.

Depending on the structure of your program, you may find the graph flashes up on the screen for such a short time that you cannot see it. This is because of the way the `Gnuplot` library works: the graph will only remain as long as the Python module that created it is still running. To insert a pause into your program then put a `raw_input()` function after the `g.plot` call. The `raw_input()` function will wait for the user to press return. A simple example is:

```
g.plot()
raw_input("Press return to continue")
```

Note we have not done anything with the result of the `raw_input()` function. The user might, for example, have typed some text but we are not interested in it and it is discarded.

If you want a print-out of the graph then use the `hardcopy()` method. This takes usually takes no parameters (although still requires the brackets). To use it simply include the line `g.plot()` (assuming your graph variable is called `g`).

So as to not waste paper we suggest you draw the graph to screen and then ask the user whether or not to print it out:

```
# Set up title/axes labels first
g.plot()    # Draw the graph on screen
answer = raw_input("Would you like to print this graph? ")
if answer == "y" or answer == "Y" or answer == "yes":
    g.hardcopy()
```

If you want to generate a *Postscript* file which can be included in a report (although note you are not expected to produce a report during this trial) then give the hardcopy method a filename as its parameter:

```
g.hardcopy("geiger_plot.ps")
```

(It is conventional to add the `.ps` extension to Postscript documents).

## 4.2 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:

```
>>> xx = array([2, 4, -11])
>>> yy = zeros(3, Int)        # Create empty array ready to receive result
>>> for i in range(0, 3):
        yy[i] = xx[i] * 2
>>> 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...

```
>>> xx = array([2, 4, -11])
>>> yy = xx + 0.1
>>> print yy
[  2.1   4.1  -10.9]
```

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

```
>>> print xx * 2
[  4    8  -22 ]
```

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

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

...but of course the arrays must be of the same dimensions (see Figure 4.1):

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

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 3.10.

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

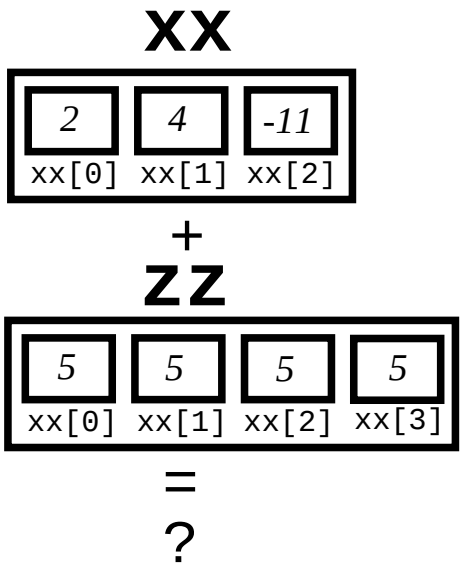


Figure 4.1: Arrays must be of the same dimensions in order to add them together

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

We create such an array in Python as follows:

```
>>> xx = zeros([2,2], Float)  # Recall 2nd parameter to zeros is the type
>>> print xx
[[0 0]
 [0 0]]
>>> xx[0][0] = 5.3
>>> print xx
[[ 5.3  0. ]
 [ 0.   0. ]]
>>> xx[1][0] = 4
>>> xx[0][1] = -10
>>> xx[1][1] = 16
>>> print xx
[[  5.3 -10. ]
 [  4.   16. ]]
>>> 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 enlcosed 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, although you will not find much need for them in the first year problems you will be attempting in this trial.

## 4.3 Functions you may need for the first-year problems

This section deals with advanced ways of reading data in from a file and how to get the “tables” you produced in Chapter 3 to line up properly.

### 4.3.1 Reading files

In some of the first-year problems you are required to read in numerical date from a file. If each line contains only one element then this is generally simple—the details are discussed in Section 3.12. However, if the file you are reading in from is itself a table, i.e. each line contains more than one datum, you will need to use the `string` library.

Begin your program with `from string import *` to have access to the functions you will need in this section. Then create a file object and read in the first line as usual. `fin` is the conventional name for a file variable that points to an input file, although you are free to call a file variable anything you would call any other variable. The data read in is just an example.

```
>>> fin = open("input.dat", "r")
>>> line = fin.readline()
>>> print line
1 5.06 78 15
```

The `line` variable is then simply a string containing the contents of the line. Assuming the individual datum on the line are separated by *white space*, i.e. any number of spaces or tabs, you now want to split the string into individual numbers; in this case 1, 5.06, 78, and 15. This is done using the `split()` function. The split function takes at least one argument: the string which it should split into a list:

```
>>> data = split(line)
>>> print data
['1', '5.06', '78', '15']
```

The `data` variable is now a list, each element of which contains each nubmer on the line. Unfortunately, each element is still in the form of a string so you will generally need to coerce them into numbers using either `int()` or `float()`. Recall lists can be referenced by offset in exactly the same way as arrays:

```
>>> x = int(data[0])
>>> print x
1
>>> y = float(data[1])
>>> print y
5.06
```

If you would like to read in the lines of a file until you reach then either use the `readlines()` method (note the plural) which will return an ordered list where each element is a string containing the contents of a line, or use the following code:

```
    line = fin.readline()
    while line != ''
        # Manipulate and use the data from the current line and then...
line = f.readline()
```

This will repeat the contents of the `while` loop until the end of the file is reached, i.e. until `line` is an empty string: `''`.

### 4.3.2 Random numbers

Some of the problems require random numbers. In order to generate a random number (float) in the range 0 to 1 import the `random` library by typing `from random import *` at the top of your program. You then have access to the `random()` function which takes no parameters, and returns a random number in the above interval.

## 4.4 Scope

The *scope* of a variable, is the region of the program in which it is accesible. Up until now, all 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 3.11. 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:

```
def addnumbers(x,y):
    spam = "LOCAL SPAM!!"                          # Local definition
    print "spam inside the function is:", spam
    return x + y

spam = "GLOBAL SPAM!!"
print addnumbers(5,10)                             # Function called here
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:

```
>>> def addnumbers(x, y):
        z = 50
        sum = x + y
        return sum

>>> z =3.14
```

```
>>> print addnumbers(10, 5)
15
>>> print z
3.14
```

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

```
>>> def addnumbers(x, y):
        global z
        z = 50
        sum = x + y
        return sum

>>> z =3.14
>>> print addnumbers(10, 5)
15
>>> print z
50
```

Note that making variables global can cause subtle problems and, if possible, should be avoided. As alluded to in the footnote in Section 3.11 you may find it better to pass the result back as a list.

## 4.5 Python differences

### 4.5.1 for loops

As mentioned in Section 3.6, 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.

```
>>> 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:

```
>>> 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:

```
>>> from Numeric import *
>>> xx = array([1,10.2,-509])
>>> 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.

## 4.6 Taking your interest further

If you've enjoyed looking at Python and would like to know more then the online repositry of all things Python-related is `<http://www.python.org>`. If you go to the Download section you can get Python for Windows (which is much the same as Python for UNIX). You can then write Python programs on your own computer.

In the Documentation section of the site there is an awful lot of bedtime reading. Of particular interest are the Tutorial and Library Reference.

# 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:

```
>>> 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 occured. 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.

## A.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:

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

## A.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. `primt 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
```

## A.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 on `=`, 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 and `input()` function. Incorrectly indenting your program might also cause `SyntaxErrors`

## A.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:

```
>>> x = 5
>>> 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    Numeric  Oxphys
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
```