Department of Physics

Introduction to Programming in Python

A Self-Study Course

(Version 2.2 – October 2009)

Table of Contents

1 Chapter 1 - Introduction

The aim of this course is to introduce you to the development of computer programs and to provide you with a working knowledge of the Python language.

In addition to everyday usage, physicists use computers for:

- Controlling apparatus and taking readings automatically – EG using LabView
- Processing data to extract conclusions
- Predicting the results of experiments
- Simulations of systems that are too complex to describe analytically

All but the first of these generally require using a computer programming language. The latter two can be described as 'Computational Physics'. There are computer 'applications' that allow you to address some of these problems – you will already have used Excel extensively. However, to obtain the flexibility to solve a particular problem the way YOU want to do it, you must write your own dedicated 'application' using a computer programming language. Thus skills in programming are extremely important to the modern physicist. They will be essential to you in later stages of your degree. This course provides you with an introduction to programming that should enable you to develop the more extensive skills required in this and later years. Such skills are also much sought after by a wide variety of employers.

There are many programming languages and much argument over which is 'best'. The skills that you acquire in this course should be easily transferable from the Python language that you will use to other languages. When preparing a course such as this, there is always a decision to be made over which programming language to use: in this case we have chosen Python, a relatively new (dating from 1990) but widely used language. Although not originally designed for scientific use, it is a powerful modern language, easy to learn and ideal for the rapid development of programs.

Python is widely used. For example, the installation scripts for many Linux distributions are written in Python, as is the BitTorrent software. Google uses Python extensively, as do AstraZeneca, the pharmaceutical company, and Industrial Light and Magic, the company behind the special effects for Star Wars and many subsequent movies.

Python is interpreted, which means that the computer runs another program to read your input and then execute your instructions. Programs written in languages like FORTRAN or C++ are compiled, meaning that the computer translates them into a more basic machine-readable form which can be executed much faster. The speed disadvantage of interpreting will not be an issue in this course. In fact, not having to compile and link your code before running it will save you considerable time. Nevertheless, you should be aware of this fundamental difference between interpreted and compiled languages. Python's high-level language features also mean that programs will typically be much shorter than equivalent ones written in C or C++.

Python can be linked to compiled code from other languages, which can largely get over the speed penalty. This is done, for example, in the NumPy and SciPy modules which we'll encounter during this course. They provide fast vector and matrix handling, as well as Python interfaces to fast routines for tasks like integration, solving differential equations and linear algebra. If you become a Python expert, you'll be able to make links to your own compiled code.

Whilst one can use Python, rather like a powerful programmable editor, from the command line, you will soon want to write many lines of program and save them to a file. You then tell Python to execute the file. For this you need an editor. You can use any ordinary text editor (NOT a work processor!!). There is a very good editor bundled with Python called IDLE which helps you with your syntax. We recommend that you use this editor.

One advantage of Python is that, although it is a complex and rich language, it is easy to learn the basics. Once you have a grounding in Python, you can extend your knowledge later. Some topics in this guide are marked thus: *Advanced Topic.* You can skip over these and come back to them later if you have time.

In this course you will be learning 'procedural programming'. That is writing a set of functions to do what you want and arranging them in files called 'modules'. There is an alternative technique called 'Object Oriented Programming' that you will hear mentioned and can read about in Hetland. That is an *Advanced Topic* and is not covered in this course.

When you are writing programs to solve Physics problems, try to keep clear the distinction between your algorithm or method of solution and the overall program. You can then easily change the algorithm function without affecting the way your main code works.

All programming languages implement the same basic constructs, loops, decision-making and so on. Once you have seen them in one language (Python) it will be much quicker to pick up a different language later if you need to.

If you are keen to learn a compiled language, either now or later in your degree, the department maintains similar courses in both Fortran and C++. These can be made available to you on request.

Note 1: Throughout this document, we have used the Courier font, which looks like this: `This is in the Courier font,` to represent Python code or input and output from the computer.

Note 2: These course notes, and the associated exercises, attempt to be both a tutorial and a reference document. DO skip the *advanced topics* on your first run through. DO look ahead to the later chapters when you need more detail on a subject.

2 Chapter 2 - Resources Required for the Course

2.1 Programming Language

Python!!

This is available on the ITS *Linux* service. Type `python` at the Linux prompt.

Python is also available for other operating systems (See below).

2.2 Computer Operating Systems

You will already be familiar with the *Microsoft Windows* operating system. It is widely used but there are others! Physicists very often use the *Unix* operating system. This comes in several forms. One of the most popular is *Linux.* As well as learning to use the Python language, you are expected to develop your programs under the *Linux* system. Knowledge of this system is required for your work in this and later years.

The Linux system is provided by the ITS. You can access it from the networked PCs in the computer classrooms or from your own computer.

A good introduction to *Linux* and to using the ITS Linux service is provided on the ITS web pages at:

http://www.dur.ac.uk/its/linux/

You are recommended to use the Vega time-saharing service provided by ITS.

There is also a very useful guide to *Linux* provided by ITS at:

http://www.dur.ac.uk/resources/its/info/guides/169linux.pdf

Although the ITS Linux service provides a window manager (rather like MS Windows) called GNOME, it is common under Linux to work much of the time using a terminal window with a command line prompt. This is something you should get used to doing.

2.3 Additional Libraries (Modules)

You are learning to program in order to use computers to do, amongst other things, Computational Physics. Python does not provide sufficient facilities for this purpose built into the language. However, adding 'modules' to Python is easy, as you will find out when you add your own! There is a large number of modules (or libraries) already available for Python. We will use just 3 of these:

- NumPy and SciPy – These provide fast vector and matrix handling, as well as Python interfaces to fast routines for tasks like integration, solving differential equations and linear algebra.
- Matplotlib or Pylab – This is a standard plotting package for drawing graphs. (Import pylab – it will import the lower level matplotlib for you).

These modules have been installed for you on the ITS *Linux* service and are available within the Enthought Python distribution (See below).

There is good documentation for NumPy and SciPy at: http://www.scipy.org/. Try the tutorial under 'Getting Started'.

Importing these or any other module into Python is easy:

EG `import numpy  # Import the standard module numpy`

Or `import mycleverlib  # Import my own module`

Not really too hard! We will return to the use of modules later in the course.

(Note: All the text after a # on each line is treated as a comment and is not executed).

2.4 Editors

You can do a lot with Python just from the command prompt within a terminal window. However, at some point you will want to save loads of Python lines as a 'program', sometimes called a 'script'. For this, you need an editor. You CANNOT use word processors (like Word) for this!! There are many suitable editors. It is best to use one that 'knows' about Python and will help you get the syntax right.

The standard editor for Python is called IDLE. Wherever you find Python you are very likely to find IDLE. It provides a simple (but rather good) editor for writing your code. IDLE also provides an interactive shell (another window) to run the code in. This can be useful but can cause confusion. It is best to run your programs directly from the command line and NOT in this window.

IDLE also comes as part of the Enthought distribution of Python (see below).

It can be accessed under *Linux* by typing `idle -n &` at a command prompt. The & makes it run in the background so that you get your *Linux* prompt back to do other things. The –n MUST be included or you will get an error. (Remember that *Linux* commands are case sensitive).

2.5 Where to do the Work

Python will run happily on most operating systems. You could do all of the course work on a *Microsoft (MS)* Windows machine or a Mac if you wish. However, you are expected, as a part of the course, to learn the basics of the *Linux* operating system. You should therefore start the course work on the ITS Linux service via the networked PCs in the classroom where the demonstrator sessions are held. (One of these classrooms is in room 140 in the Physics department and you can use this room any time when it is NOT in use for a class).

You can log in to the ITS Linux service from your own PC using a VNC client. A free version of such a client (for the MS Windows operating system only) is available from:

http://www.realvnc.com/products/download.html

This is only possible if you connect your computer directly to the Durham University system (EG In colleges). It will NOT work from outside the University. Access can be gained from outside the University using 'Putty'.

Details of how to access the Vega service are given by ITS at:

http://www.dur.ac.uk/its/linux/remote_access/

You are encouraged to transfer work to your own PC or laptop in order to continue to work anywhere. Whether your PC runs MS Windows, MacOS or Linux, you can download an excellent version of Python, including all of the additional modules you need for the course, from 'Enthought'. There are free academic versions at:

http://www.enthought.com/products/epddownload.php

2.6 Books

You don't need a book to learn Python! Just sit at a computer and write programs. Having said that, most programmers keep a good Python book nearby for reference. The first port of call however is the web. The Python documentation is very extensive. See: http://www.python.org/doc/. This is the font of all knowledge for Python. (Be sure to use the documentation for version 2.x).

You can download much of this to your computer should you wish.

We recommend two books:

- Beginning Python by Magnus Lie Hetland. This is an excellent introduction to the language with loads of examples
- Python in a Nutshell by Alex Martelli. This is more of a handy reference book

If you just want one good book to use, get 'Beginning Python'. We will refer to it in these notes as 'Hetland'. The course largely follows the early chapters of Hetland. In fact, chapters 1 to 11 (but not 7 and 9 which cover object oriented techniques) match the course well. However, Hetland does **not** cover scientific computing using NumPy and SciPy which is included in this course.

Note: There is a useful reference for much of the more commonly used aspects of Python in Hetland in appendix B.

3 Chapter 3 - Getting Started

Before you start the course, you may also like to put Python on your computer and try the Python tutorial online at:

http://docs.python.org/tut/tut.html

(Try section 3 first).

The great thing about Python is how fast you can get started. You may like to try it first in the familiar environment of MS Windows before moving to Linux.

As you learn the language, ALWAYS have a Python prompt available. If you learn something new, try it at once!

Log on to the ITS Linux service and bring up a terminal (one of the icons on the bar at the bottom of the screen). At the terminal prompt, type `python`. You will get a new prompt in the form: `>>>`. The Python interpreter is now running.

Note: Much of this chapter is covered in more detail in chapter 1 of Hetland.

3.1 Numbers

Before we write any programs, we will start by using Python from the command line. It is rather like using a fancy calculator. Even after you have written a load of code, it is still very useful to quickly try things out at the command line.

Try this example:

```
>>> (2+4+6)*3-12/3
32
>>> 8**2 # ** is used for power 2.
64
>>>
```

It works! Note the use of a # to introduce a comment. The interpreter ignores all of the rest of the line after the #.

These were integers. Let's try it with real numbers:

```
>>> (2.0+4.0+6.0)*3.0-12.0/3.0
32.0
>>> 8.0**2
64.0
>>>
```

This looks fine. But now try:

```
>>> 7.0/4.0
1.75
>>> 7/4
```

```
1
>>>
```

Not so fine! The lesson is that computers distinguish between exact integer arithmetic and real number (or floating point) arithmetic. In particular, whenever dividing two integers does not result in an integer, the result is truncated (rounded down) to the nearest integer. If you mix integer and floating point numbers, they are all converted to floating point by Python:

```
>>> 34+1.0
35.0
>>>
```

You can also convert from integer to float and back again, but note that conversion to integers is done by *truncating*

```
>>> float(1)
1.0
>>> int(3.678)
3
>>>
```

Python will do lots of auto converting for you which is very convenient. But beware – sometimes it will not do exactly what you wanted. It is good practice, if you want a floating point number to always include the decimal point. Thus `12.3+1.0` is better style than `12.3+1`.

Advanced Topic:

If you would like Python to NOT use integer division but to convert for you and give the answer you might expect, you can tell it to do so. How to do this is explained in Hetland chapter 1:

```
>>> from __future__ import division
>>> 7/4
1.75
>>> 7//4  # Force python to do proper integer division
1
>>>
```

3.2 Assignments, Strings and Types

You can assign values to **variables.** The computer will save the value in some memory pointed to by the variable. If you do this, you won't see the value printed on the screen unless you ask for it. If you just type the variable name, Python will return the value (or you can use the print command:

```
>>> x=3
>>> x
3
>>> y=2
```

```
>>> z=7
>>> x*y*z
42
>>> a=x*y*z
>>> print a
42
>>>
```

Remember that the = sign is used for *assignment* in Python. This is NOT the same as its use in maths. Thus `x=x+1` makes perfect sense. It means increment the value of the x variable by 1. It is NOT a maths equation!

Variables can be of types other than integer and float. For example, they can be strings of characters; in other words text. A string is a special case of a list object as we will see later. String handling is easy in Python. It is well explained in chapter 3 of Hetland.

A string is always enclosed in either single or double quotes (But not a mixture of both!).

Strings can be added (concatenated or joined):

```
>>> "spam"
'spam'
>>> "spam," + " " + "eggs and spam"
'spam, eggs and spam'
>>>
```

Strings can be duplicated by multiplying them by an integer:

```
>>> "Spam, "*4 + 'beans and spam'
'Spam, Spam, Spam, Spam, beans and spam'
>>>
```

Python does not require you to be strict about what type a variable is (the language is 'weakly typed'). However, this is sometimes important to a Physicist. If you must, you can find the type of a variable thus:

```
>>> a=1; b=1.0; c='1.0'
>>> type(a)
<type 'int'>
>>> type(b)
<type 'float'>
>>> type(c)
<type 'str'>
>>> print a,b,c
1 1.0 1.0
```

```
>>>
```

You can convert a string to a number (within reason) and visa versa:

```
>>> str(12.3)
'12.3'
>>> float('12.3')
12.300000000000001
>>>
```

Note that we didn't quite get what we expected from the `float` function. The computer has a limited precision. It stores the nearest number to what you want within its ability. We will come back to this issue of *precision*.

Note that you can put several short statements on the same line separated by semi colons. It is very poor style to do this for more than very simple lines as above.

3.2.1 A First Mention of Functions

The use, and creation, of functions will be handled later in this course. However, we have already mentioned the `str()` and `float()` functions above.

Python will provide you with a huge number of functions that are already written. Some of these are 'built-in' to the language like those above. Others are provided by modules that you 'include'. We will discuss functions in detail later on. For now, just note the format of a function call:

result = function(arguments)

For example, we can use 2**3 to raise 2 to the power 3. This is really just a shorthand for the function `pow()`:

```
>>> 2**3
8
>>> pow(2,3)
8
>>>
```

Note 1: The use of brackets of type () for function calls. Other shapes of brackets mean different things in Python. *Don't muddle them up!* We will come across [] and {} later on.

3.2.2 A Brief Mention of Methods

We will make brief mention of 'methods' here and there. Methods are a part of Object Oriented Programming which we are not considering in this course. We will use functions throughout the course rather than methods. However, you will need to use some methods that are provided by Python so you must learn the syntax.

They can be thought of as an alternative way of calling some already-written code and take the form:

Result = object.method(arguments). Notice the . (dot) in the syntax.

An example is appending to a list (We will talk more about lists in section 6.1):

```
>>> a=[0,1,2] # Make a list
>>> a.append(3) # Use the append() method
>>> print a
[0, 1, 2, 3]
>>>
```

3.3 Complex numbers (*Advanced Topic)*

This is a type not often needed by an everyday Python user but much beloved of Physicists!

Python supports complex numbers. They are written in the form `3+5j`, corresponding to `3+5i` in usual mathematical notation. Here are some simple examples using complex numbers:

```
>>> d=3+5j
>>> type(d)
<type 'complex'>
>>> d.real
3.0
>>> d.imag
5.0
>>> abs(d) # absolute value or magnitude or modulus
5.8309518948452999
>>> e=1+1j
>>> print d*e
(-2+8j)
>>>
```

Note1: We used `d.real` to get the real part of a complex number. This is a way of getting an 'attribute' of the variable. It is a first hint of what is called 'object oriented programming'. This is not a part of this course but will be mentioned now and again. For now, just remember the syntax.

Note 2: There is no separate type for imaginary numbers in Python. They are treated as complex numbers whose real component is zero.

3.4 Errors and Exceptions

By now, you will have made some mistakes and will have seen your first Python 'Traceback'. Be aware that errors are referred to as 'exceptions' in Python.

For example:

```
>>> float('gumby')
Traceback (most recent call last):
  File "<pyshell#90>", line 1, in <module>
    float('gumby')
```

```
ValueError: invalid literal for float(): gumby
>>>
```

When an error occurs in a program, it may occur in a function that is called from another function in another module etc. The traceback gives very valuable information about the error that occurred AND where it occurred (eg the line number in a file).

In this case, the error was at the Python prompt (the Python shell). We supplied a string to the function `float()` which it really can't convert.

You may also get syntax errors. This is where you have written something that Python simply doesn't understand. For example:

```
>>> str(1.23a)
SyntaxError: invalid syntax
```

Now let's look at a more detailed traceback:

```
Traceback (most recent call last):
  File "\\elite\home\nad\My
Documents\AllDocs\teaching\Computing\2009\TestPrograms\test01.
py", line 7, in <module>
    T.theTest(theData)
  File "\\elite\home\nad\My
Documents\AllDocs\teaching\Computing\2009\TestPrograms\testMod
ule.py", line 4, in theTest
    y = x.fred() # Cause an error
AttributeError: 'numpy.ndarray' object has no attribute 'fred'
```

The traceback starts with what was happening at the top level. An error occurred in my main program (called <module>) at line 7. I am told the name of the file and the line number. However, there is more… The actual error was in another module file (name given) that contains the function called 'theTest()' that I called from my main program. The line that caused the error is identified and the cause of the error is given. The function 'fred()' does not exist (No such attribute). The error descriptions become clearer as you get to know the terminology.

These tracebacks provide you with the main information you need to 'debug' your program. IE Find out what went wrong. Debugging is discussed further in chapter 9.

3.5 Precision and Overflow

As physicists, we are very often using floating point numbers (real numbers) to do calculations. There are some pitfalls with using floats of which you should be aware.

3.5.1 Precision

Computers have a limited dynamic range for storing numbers and also, for real numbers, a limited precision. This can be a significant problem in computational physics where the 'rounding errors' in real numbers may add up to a large error in your result.

As a trivial example:

```
>>> 1.0/3.0
0.33333333333333331
```

What is that 1 doing on the end?

Python has done its best but cannot give the answer to an infinite number of decimal places. The precision of Python floating point numbers is about 1 part in 10^{16}.

3.5.2 Overflow – Large Numbers

Python can store really large numbers. At a certain point it will automatically start using `long integers.` These require more storage.

Try:

```
 >>> 1000000*1000000
1000000000000L
>>>
```

The L on the end shows that Python has used a long integer but you don't really need to know.

For real numbers, Python can store enormous numbers. It will eventually give an error when the exponent is just too large:

```
>>> 1e200**2
Traceback (most recent call last):
  File "<pyshell#119>", line 1, in <module>
    1e200**2
OverflowError: (34, 'Result too large')
>>>
```

Note 1: We use ** to mean 'raise to the power'

Note 2: We can use 1e200 to represent the number 10^{200}.

3.6 Getting Help

There is a LOT of online help for Python. The full documentation set for Python is online at:

http://www.python.org/doc/.

The IDLE editor will prompt you with popup 'tips'. These are very useful for remembering for example the parameters required by a function call.

There is a built-in help system. Use the function `help()`. Enter a Python keyword or function name between the brackets. Try just typing `help()` at the Python prompt to get started.

4 Chapter 4 - Input and Output (IO)

This chapter describes how to get data in and out of Python programs. Section 4.1 describes input and output to/from the screen and contains much of the detail that you need to know about IO, such as the % format specifier.

You will also want to save and load data to/from data files. This is described in detail in Chapter 11. However, there is excellent support for simple file IO of numbers within the NumPy module. This technique is described here in section 4.2 so that you can do file IO without reading the gory details of chapter 11.

4.1 Screen Input/Output

4.1.1 Output

We have already seen how to get output from the command line. Generally, within a program, we use the `print` command to output things to the screen. If you don't tell Python what format to use on output, it will decide for you:

```
>>> a=1.123456789
>>> print a
1.123456789
```

- but you can tell Python exactly how to write to the screen using a format conversion specifier (%):

```
>>> print 'a= %.2f' % a, 'cms'
a= 1.12 cms
>>>
```

To output more than one thing, we separate them with commas.

4.1.2 The Format Conversion Specifier

The % sign in the above example is used to tell Python what format to use to output to the screen. The details of how to use % are in chapter 3 of Hetland. Here we have specified a float format (f) with 2 decimal places (.2). The % in the string tells the `print` function that a format specifier is coming next. There must be one format specifier for each item you output:

```
>>> a=1.23
>>> b=123.45
>>> print 'a = %5.2f' % a, 'b = %.4e' % b
a =  1.23 b = 1.2345e+002
```

The format specifier %5.2f means: Output the variable as a floating point number with a total field width (all digits plus the decimal point) of 5 and with 2 decimal places.

We can get a similar result like this:

```
>>> a=1.23
>>> b=123.45
>>> print 'a = %5.2f\tb = %.4e' % (a,b)
```

```
a =  1.23 b = 1.2345e+002
```

Here the string contains two conversion specifiers and the two variables have been put at the end as the tuple: `(a,b)`. (A tuple is just a list of variables – see section 6.1.5). The \t special character used here is described in section 4.1.3.

There are quite a lot of conversion specifier types. The most useful are:

- f or F – Floating point (EG 1.234)
- d or I – Integer (EG 123)
- e or E – Exponential format (EG 1.234e+002)

4.1.3 Special Characters in Output

There are some special characters that it is useful to output to the screen. They are represented within a string by a backslash followed by a character. The most useful examples are:

- \n – This inserts a 'carriage return'
- \t – This inserts a tab

We used the \t special character in the above example to insert a tab character.

Note: If you actually want the backslash character in your output string, you must enter it twice.

4.1.4 Input

Sometimes you will want the user of your program to type in some input from the terminal. You can 'prompt' him to do so by outputting a string and then wait for his input. To do this, there is an `input()` function. This can be used to input directly to a Python variable.

```
>>> a=input("Input a number: ")
Input a number: 123.45
>>> print a
123.45
>>>
```

This is fine for the input of numbers but not great for the input of strings. The problem is that `input()` assumes that what you type is valid Python.

```
>>> b=input("Input your name: ")
Input your name: Ron
Traceback (most recent call last):
  File "<pyshell#2>", line 1, in <module>
    b=input("Input your name: ")
  File "<string>", line 1, in <module>
NameError: name 'Ron' is not defined
>>>
```

To avoid this error, the user must enter the string with quotes around it thus:

```
>>> b=input('Input your name: ')
Input your name: "Ron"
>>> print b
Ron
>>>
```

To avoid this, you can use the function `raw_input()`. (See Hetland Page 26). Doing this however means that your input is **all** treated as a string and, if you want another type, you must convert it:

```
>>> a = float(raw_input('Type a number: '))
Type a number: 23
>>> print a
23.0
>>>
```

Which input function you use is up to you.

The `raw_input()` and `input()` functions will use the string that you give it as a parameter as a 'prompt'. It writes this on the screen and then waits for some input terminated by a 'carriage return' (The 'enter' key).

Note: You can apply a function to the result of another function directly as shown above. `raw_input()` returns a string that is the input to the function `float()`.

4.2 File Input and Output

If you need to write a lot of complicated mixed text and numbers to a file, you need to understand how file IO is done in Python (Chapter 11). However, saving or loading a NumPy array can be done very easily.

This section makes use of quite a few concepts that you have not met yet. Just try this example so that you can see how easy it is to input and output data to file. Before we start, what is a NumPy array? There is an introduction to NumPy in section 6.3. Take a quick look now or just try the example and learn more about NumPy later.

4.2.1 Saving an Array to File

The NumPy module provides the `savetxt()` function to do this. Just get your data into an array of suitable size and shape and this function will write it to file for you. You should only do this with NumPy arrays. If you try to read from a file that is a mixture of types, NumPy will do its best but may give you confusing results.

Here is an example. The first line loads (imports) the NumPy module so that you can use it:

```
>>> import numpy as N
>>> data = N.zeros((5,2), dtype='float') # Make an empty 2-d
array
>>> data[:,0] = N.arange(5.0) # Fill one column with numbers
>>> data[:,1] = data[:,0] ** 2.0 # Fill second column with
squares
```

```
>>> numpy.savetxt('myFile.txt', data, fmt='%.3f') # Save to
file
```

Try all this from the command line. Then look at the text in your file. It should look like this:

```
0.000 0.000

1.000 1.000

2.000 4.000

3.000 9.000

4.000 16.000
```

Note that savetxt() has used a blank space between entries in each row. You can change this (if you must) using the delimiter parameter. For example:

```
>>> numpy.savetxt('myFile.txt', data, fmt='%.3f',
delimiter=',')
```

will use a comma as the delimiter.

4.2.2 Loading an Array from File

Loading the data back from your file to a NumPy array is equally simple using `loadtxt()`. Be sure to tell `loadtxt()` what the delimiter is (or it will assume a blank space):

```
>>> import numpy as N

>>> data=N.loadtxt('temp.txt')

>>> print data

[[  0.   0.]

 [  1.   1.]

 [  2.   4.]

 [  3.   9.]

 [  4.  16.]]
```

5 Chapter 5 - Programs (Scripts)

If you already know how to create a program in a file using an editor and run the program, you can skip this section, but do the exercises anyway.

5.1 My First Program – 'Hello world'

Before we go any further, its time to write a few 'scripts' or programs.

Do this with the IDLE editor on the Linux system. Type your program into the editor, save it as a file then run the file

The first program that everyone writes in every computer language is the 'Hello world' program. This just writes 'Hello world' on the screen when it is run. This can take 10 or even 20 lines to do in a language such as C++ or Java. In Python, it is 1 line. So let's do it…

- Log on to the Linux system
- Open a terminal (Click on the relevant icon)
- At the Linux prompt type: `idle –n &`
- Select File -> New window to get the editor window open
- Type in: `print "Hello world"` followed by a carriage return
- Select File -> Save as… Give it a name (with .py extension) and click Save
- The file will be saved in the directory from which you ran IDLE
- In the Linux window, type: `ls` and check that your program file is there
- Type: `python myprogram.py` to run your program (With your own program name)
- It should write `Hello world` to the screen.
- Congratulations! You wrote a working program.

Note 1: All Python program files should have the .py extension. EG myprogram.py

In all sections from now on, we provide exercises for you to try. Model programs are provided on DUO for all of these exercises. However, do write your own first then look at the model solution.

5.2 Exercises

5.2.1 Exercise 5.1

Write a program that evaluates the function:

$y = x^2-1$

at a single point. The program should request a value of x from the screen, print out the equivalent value of y and stop.

Suggestions:

Put a comment at the top of the program. This comment at the top of a program file is special. Enter it as a string (IE In quotes) rather than using a #. You will see why later.

Make the program work with floating point numbers – much more useful than just integers.

Print the result to say 3 decimal places. (Use the % format descriptor).

Test it with various numbers. What happens if you don't do as you are asked and enter a string instead of a valid number?

Model solution is in file: Exercise5.1.py on DUO.

5.2.2 Exercise 5.2

Modify your program so that it will evaluate any quadratic: $y = ax^2 + bx + c$

You will need to ask for the coefficients from the screen.

Model solution is in file: Exercise5.2.py

Enter the coefficients used in the previous example: a=1, b=0, c=-1 and check that you get the same answer using both programs.

Model solution is in file: Exercise5.2.py on DUO.

6 Chapter 6 - Sequences, Lists and Strings

Suppose we want to evaluate the function in exercise 5.1 at more than 1 point and save all of the results? This is just one example of where you might need a list or array. We will soon see that what physicists really want much of the time is arrays that represent vectors and matrices. These are just a special sort of list and we will meet them later.

Strictly speaking, Python provides various sorts of 'sequences' but the most important sequences within Python are lists and strings. The most important sequences for physicists however are NumPy arrays. These are introduced here.

So…Lists and strings are useful but don't spend too much time on them. All of the physics will be done using NumPy arrays.

Note 1: You can have 2-d lists but, if you are handling numbers, NumPy arrays are much better.

Note 2: You will come across a sequence called a 'tuple' (section 6.1.5). It is just a list whose elements cannot be changed.

6.1 Lists

We will look briefly here at how lists are created and used. For more detail, see chapter 2 of Hetland.

You can create a list of objects. The objects don't have to be all of the same type and could even include other lists. We use square brackets [] for lists:

```
>>> a=[3,54,26,90]
>>> print a
[3, 54, 26, 90]
>>> b=[3,54,'llama',a] # Include the list a in the list b
>>> print b
[3, 54, 'llama', [3, 54, 26, 90]] # Note the nested brackets
>>>
```

The elements of a list are numbered from zero and you get an error if you ask for an element that is out of range. It is a very common error to forget that the *elements of a list start at zero* (not 1). So the list above called `a` has 4 elements numbered 0,1,2 and 3. The address of an element in a list is known as its 'index'.

You access one or more elements of a list by giving the index(s) of the element(s) you want:

```
>>> a=[3,54,26,90]
>>> b=[3,54,'llama',a]
>>> print b
[3, 54, 'llama', [3, 54, 26, 90]]
>>> a[0]
3
```

```
>>> b[3]
[3, 54, 26, 90]
>>> b[3][2] # Interpret as (b[3])[2]
26
>>> b[4]
Traceback (most recent call last):
  File "<pyshell#40>", line 1, in <module>
    b[4]
IndexError: list index out of range
>>>
```

Note 1: You can access elements of a list from the right hand end using negative indexes.

Note 2: You can get the length of a list using the `len()` function.

Note 3: You can change any element of a list.

In the list called `a` above:

```
>>> a[-1]
90
>>> a[-2]
26
>>> len(a)
4
>>> a[2] = 'gecko'
>>> print a
[3, 54, 'gecko', 90]
>>>
```

6.1.1 Slicing Lists

You can get a 'slice' of a list. This is a very important technique which is also used for NumPy arrays. You will often want to access just one part of a list or array. To do this, use two indices (start and end) separated by a colon. Note that the 'end' index actually refers to the element *after* the last one you want!!

```
>>> a=[12,23,34,45,56,67,78]
>>> print a[3:6]
[45, 56, 67]
>>>
```

If you miss out one of the indices, Python will assume you want all the rest, from the beginning or up to the end:

```
>>> a[3:]
```

```
[45, 56, 67, 78]
>>> a[:3]
[12, 23, 34]
>>>
```

6.1.2 2-d Lists – *Advanced Topic*

You can have 2-d lists – it's just a list of lists! - but, especially when manipulating numbers, it is *far better to use NumPy arrays* (see section 6.3).

Addressing a 2-d list is done thus:

```
>>> a=[[0,1,2],[3,4,5],[6,7,8]] # Create a 2-d list
>>> print a
[[0, 1, 2], [3, 4, 5], [6, 7, 8]]
>>> print a[1][2] # Addressing a 2-d list
5
```

6.1.3 Basic List Operations

There are lots of operations that can be performed on lists. Here we mention just a few of the most useful:

You can *add* one list to another:

```
>>> a=[1,2,3]
>>> b=[4,5,6]
>>> c=a+b
>>> c
[1, 2, 3, 4, 5, 6]
>>>
```

You can *append* a new value to the end. This is one of the most useful features of a list. You cannot append to a NumPy array.

```
>>> a=[1,2,3]
>>> a.append(4) # Note the dot!  This is a method.
>>> print a
[1, 2, 3, 4]
>>>
```

You can *insert* a new value in the middle of a list:

```
>>> a=[1,2,3,4,5,6,7,8,9]
>>> a.insert(3,99)
>>> print a
[1, 2, 3, 99, 4, 5, 6, 7, 8, 9]
>>>
```

You can look to see if a list contains a particular value. For this we use the membership operator known as *in*:

```
>>> a=[1,2,3,4,5,6]
>>> 3 in a # Think of it as a question: Is 3 in a?
True
>>> 7 in a
False
>>>
```

What you get back is a Boolean or logical value. This can take only the value True or False. The `in` operator is most useful in loops which we will look at later.

6.1.4 Fancy List Handling – zip() and map() – *Advanced Topic*

You can do some very clever things with lists. In general however, the 'array' type that we introduce later in the course (in module NumPy) is more useful to physicists.

Have a look at appendix B of Hetland where he has a summary of built in functions and methods. I will give just two examples here:

Example 1. Try the `zip()` function:

```
>>> a=[1,2,3]
>>> b=[4,5,6]
>>> c = zip(a,b)
```

Now print c and see what zip() has done to your 2 lists.

Example 2. Try the `map()` function:

You may have a list of integers and want to make them all into floats. Python won't let you do this in case the list contains types that cannot be converted. However, the map() function allows you to 'map' any function onto all the elements of a list. Of course, if the list contains unsuitable elements, you will get an error:

```
>>> a=[1,2,3,4,5]
>>> b=map(float,a)# Apply the float() function to all elements
>>> print b
[1.0, 2.0, 3.0, 4.0, 5.0]
>>> a.append('fred')
>>> print a
[1, 2, 3, 4, 5, 'fred']
>>> b=map(float,a)
Traceback (most recent call last):
  File "<pyshell#16>", line 1, in <module>
    b=map(float,a)
ValueError: invalid literal for float(): fred
```

6.1.5 Tuples

Tuples are lists that cannot be changed. In general, don't worry about them, just use lists! However, they can be useful and you will come across them.

If you use a library function that happens to return a tuple, just treat it like you would a list but *don't try to change it*! Tuples use round brackets for assignment and display:

```
>>> a=(1,2,3)
>>> a
(1, 2, 3)
>>> type(a)
<type 'tuple'>
>>> print a[1]
2
>>> a[1] = 99
Traceback (most recent call last):
  File "<pyshell#88>", line 1, in <module>
    a[1] = 99
TypeError: 'tuple' object does not support item assignment
>>>
```

6.2 Strings

We have already come across strings. They are very like a list in which every element is a character. You can access their individual characters by indexing or slicing.

There are some differences to lists. You cannot, for example, change the elements of a string:

```
>>> a='This parrot is dead'
>>> print a[5:11]
parrot
>>> a[5] = 'f'
Traceback (most recent call last):
  File "<pyshell#100>", line 1, in <module>
    a[5] = 'f'
TypeError: 'str' object does not support item assignment
>>>
```

Python is very good at handling strings. There is a rich set of methods to do things to them.

See chapter 3 of Hetland for details.

A few useful examples are:

`find()`: Find a substring within a larger string

`split()`: Splits a string up into its elements into a list.

`join()`: Joins a list of strings (The opposite of split)

For example:

```
>>> a="Monty Python's Flying Circus"
>>> a.find('Monty')
0
>>> a.find('Python')
6
>>> a.find('Gumby')
-1
>>>
```

Note 1: If you want to use a quote in a string (EG for an apostrophe), you must use the OTHER type of quotes around the string

Note 2: `find()` returns the index where the substring starts or -1 if the substring is not found.

3.3 Numpy arrays – An Introduction

Whilst Python lists can be very powerful, what the physicist really needs is arrays that can be used to represent vectors and matrices. Python was not specifically designed for physics so it has no such arrays. Nor does it have much in the way of mathematical functions. All of this is added to the language by importing an essential module called NumPy.

We have not yet covered the writing of functions or the importing of modules (which are essentially just a collection of functions). However, the arrays and functions provided by NumPy are so useful for solving physics problems that we will take a first look at them now.

More detail on writing functions and using modules are given in chapter 8. More details of the NumPy module are given in chapter 10.

There is an excellent introduction to NumPy (and its partner module SciPy) on the web at:

http://www.scipy.org/Getting_Started

Have a quick look at it now. However, some of the detail and examples in that introduction will mean more when you have looked at the later chapters of this course.

Arrays are very like lists. The most important differences are:

- All of their elements must be of the same type
- Their 'shape' can be changed (EG from 1-d to 2-d)
- Multi-dimensional arrays are addressed differently to lists
- You can't append to an array. Once its length is set, it's fixed.
- Manipulating arrays is much faster than manipulating lists

For the technically minded, NumPy is implemented in the C language and is a compiled library. All Python list handling is interpreted and thus much slower. Thus using NumPy is the secret of doing complex physics problems in a slow language (Python) very quickly.

For solving physics problems, if you are in doubt which to use, *use arrays, not lists.*

6.3.1 Using NumPy

The process of adding a module (or library) to Python is very simple and is called 'importing'.

At the Python prompt, type `import numpy` (Note the lower case name).

This will, as if by magic, add a vast number of functions to Python. If you want to see how many, try typing `dir(numpy)`. To use them, just prefix their names with the name of the module.

EG One way to create an array is using the NumPy function called `arange().`

Try this:

```
>>> a=numpy.arange(9, dtype='float')
>>> print a
[ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.]
>>>
```

We have specifically told NumPy that we want floats using the `dtype` parameter. If you don't specify what you want you will get the default *on your system*. This may be integers or floats. You can also tell arrange you want floats by using a float as an argument:

```
a=numpy.arange(9.0)
```

You can change this 9 element 1-d vector into a 3x3 matrix thus:

```
>>> a.shape=(3,3) # Use the shape method
>>> print a
[[ 0.  1.  2.]
 [ 3.  4.  5.]
 [ 6.  7.  8.]]
>>>
```

You can manipulate an array just like a variable. You can't (easily) do this with a list:

```
>>> b=a**2
>>> print b
[[  0.   1.   4.]
 [  9.  16.  25.]
 [ 36.  49.  64.]]
>>>
```

6.3.2 Addressing and Slicing Arrays

Addressing and slicing arrays is done in a very similar way to lists. We use the same square brackets: []. So, in 1-d:

```
>>> a=numpy.arange(9, dtype='float')
>>> print a
[ 0.  1.  2.  3.  4.  5.  6.  7.  8.]
```

```
>>> print a[1:5] # A slice
[ 1.  2.  3.  4.]
>>>
```

Just as for a list, we have taken a 'slice' from the array, `a[1:5]`. Remember that the slice starts at index 1 but ends at index 4. IE One less than the last index given.

The difference comes with arrays of more than one dimension. So in 2-d:

```
>>> a=numpy.arange(9,dtype='float')
>>> a.shape=(3,3) # Make it 2-d
>>> print a
[[ 0.  1.  2.]
 [ 3.  4.  5.]
 [ 6.  7.  8.]]
>>> print a[2,1] #Print one element of the matrix
7.0
>>>
```

We have addressed one element of the matrix at row 2, column 1 as: `a[2,1].`

Note that we address a 2-d NumPy array with the 'y' coordinate first, then the 'x' coordinate. So think of the element that you want from your matrix 'a' as having an address a(y,x).

This form of addressing, with a comma, is *not* valid for lists.

This section was just a taster. We will return to arrays in chapter 10. Remember, if you are manipulating sequences of numbers, especially in more than one dimension, *use arrays not lists*.

6.4 Dictionaries – *Advanced Topic*

Dictionaries are a very powerful type in Python. However, they are not essential for this course and are not covered in any detail here. They are covered in chapter 4 of Hetland.

They are another form of sequence, like a list, but the order of the elements is not fixed. Thus there is a *key* for every element. They use curly brackets like {}. They can be used to form a sort of mini database. For example, you could store your friend's 'phone numbers in a dictionary:

```
>>> phonebook = {}
>>> phonebook['Fred'] = '12345'
>>> phonebook['Claire'] = '0234 7432'
>>> print phonebook
{'Claire': '0234 7432', 'Fred': '12345'}
>>> print phonebook['Claire']
0234 7432
>>> phonebook.has_key('Fred')
True
```

```
>>> len(phonebook)
2
```

The entries in a dictionary are *key/value* pairs separated by commas. Both the key and the value can be of most types so the 'phone numbers could have been integers. The key and the value are separated by a colon.

You can add an entry that is not already there. There is no fixed index for the elements. There is no need for an 'append' method.

Note: the method `has_key()` is used to check if the dictionary has such a key in it. If you try to access a non-existent key, you will get an error. Check it exists first.

```
>>> print phonebook['Clive']
Traceback (most recent call last):
  File "<pyshell#118>", line 1, in <module>
    print phonebook['Clive']
KeyError: 'Clive'
>>>
```

6.5 Exercises

6.5.1 Exercise 6.1 - Lists

Write a small database and some code to query it using lists.

Create a list with the names of 10 'friends'. Create a second matching list with their years of birth. Ask the user to input a name. Tell him the place (index) of that person in the list, how many friends he has in total and the year of birth of the person. The input and output should look like this:

```
Enter a name: Will
You have 10 friends
Will is number 8 in your list
Will was born in 1991
```

Try entering a name that is not in the list. The program will crash. We will look at how to handle such conditions later.

Model solution is in file: Exercise6.1.py on DUO.

6.5.2 Exercise 6.2 – Arrays

Create a 10x10 matrix of integers starting at 1 and ending at 100 as a NumPy array. The first row is the integers 1 to 10, the second 11 to 20 etc. (Use numpy.arange() and the shape() method). Print the matrix to the screen.

Now slice out the column from the matrix where all of the numbers end in the digit 5 (the 5th column). Print this to the screen. It should look like this:

```
[ 5 15 25 35 45 55 65 75 85 95]
```

Now slice out a sub-matrix from the original matrix to look like this:

```
[[35 36 37 38]
```

```
 [45 46 47 48]
 [55 56 57 58]]
```

Print it to the screen.

Now convert the sub matrix to a vector to look like:

```
[35 36 37 38 45 46 47 48 55 56 57 58]
```

This last stage is best done using the flatten() method. If you try to convert the sub-matrix to a vector using the shape() method you will run into problems. You have to make a *copy* of the sub-matrix first and flatten() does this for you.

It is important when using arrays to be aware of when you need a copy. Python will try to not make a copy if it can avoid it because it wastes memory and is slow. In our example of making a sub-matrix, it will just point to parts of the original matrix.

Model solution is in file: Exercise6.2.py on DUO.

6.5.3 Exercise 6.3 – Dictionaries - *Advanced Topic*

The program for exercise 6.1 would have been better done with a dictionary rather than two lists. Repeat the exercise but using a single dictionary to hold the data. Which of the outputs from example 6.1 cannot be generated from data stored in a dictionary and why? If you enter a friend who is not in the list, the program will crash with a different error to the version that used lists. Both errors are worth looking at.

Model solution is in file: Exercise6.3.py on DUO.

7 Chapter 7 - Conditionals and Loops

Conditionals allow a program to make decisions. They are often called 'if/then' or 'if/then/else' for obvious reasons.

Loops allow the computer to loop over a set of data. You might want to evaluate a function at 100 different values for example.

Conditionals and loops are dealt with in chapter 5 of Hetland.

7.1 Conditionals

You will often want to execute a different bit of your program depending on some condition or other. This is where the Boolean or logical variables come in. You can decide whether to execute some code depending on whether a condition is True or False.

Now that we have started writing programs, and there are quite a few lines of code involved, the examples should be typed into the editor, saved as a file and run. You therefore won't see the Python prompt (>>>) in the examples any more. Type this example in and try it:

```
a = float(raw_input('Please enter a positive number: '))
if a < 0:
    print 'That is a negative number'
    print 'That is not what I asked for'
else:
    print 'Thank you'
    print 'That is a positive number'
```

Note 1: The < symbol means 'is less than'.

Note 2: You execute a block of code lines after the `if`. Python knows how much to do because there is a semi colon (:) after the `if` statement *and* the block to be executed is *indented*. This indenting is done for you by the IDLE editor but can be put in by hand. Any number of spaces (or a tab) is an indent. Lines within the block to be executed *must all have the same indentation*.

Beware: Getting the indenting wrong is a very common error in Python.

You can test for more than one condition at a time using `elif` (short for else if):

```
a = float(raw_input('Please enter a positive number: '))
if a < 0:
    print 'That is a negative number'
    print 'That is not what I asked for'
elif a> 0:
    print 'Thank you'
    print 'That is a positive number'
else:
```

```
    print 'That looks like zero to me'
```

You can have conditionals within conditionals (an `if` within an `if`) but **be careful** to get the indentation right.

The comparison operators are not all so obvious as < and >. Here is a table of them:

Expression	**Description**
x = = y	x equals y
x < y	x is less than y
x > y	x is greater than y
x >= y	x is greater than or equal to y
x <= y	x is less than or equal to y
x != y	x is not equal to y
x is y	x and y are the same object
x is not y	x and y are different object
x in y	x is a member of the sequence y
x not in y	x is not a member of the sequence y

Note 1: It is a very common error to use x=y to test if x is equal to y (Rather than x = = y). This usually gives a syntax error. There are situations where it will not! Python will just do as it is told and set x equal to y - VERY confusing!

Note 2: The last 4 are less used but quite interesting. Try using them!

There is also an `and` operator and an `or` operator. These are used to test more than one condition at the same time:

```
if a < -10 or a > 10:
     print 'Magnitude is too large'


if a < 0 or b < 0:
     print 'One of the values is negative'
```

Note: These examples are code fragments. They won't work on their own.

7.2 Loops

Let's go back to the example of evaluating a function. We may want to evaluate it for 100 values of x and save all of the values somewhere. (Later, we will also draw a graph of it).

There are two sorts of loops in Python that allow you to execute a block of code many times: `for` loops and `while` loops.

7.2.1 Loops - The 'while' loop

These do exactly as they say. They will execute a block of code while some condition is true:

```
prompt = 'Please enter a positive number less than 10: '
x=-1 # Initialise x to an invalid number
while x<=0 or x>9:
    x = float(raw_input(prompt))
print 'Thank you. That number is fine.'
```

Try this. The while loop will repeat the indented line until the user inputs a valid number.

Note: We put the prompt string into a variable so that we can more easily change it.

7.2.2 Loops – The 'for' loop

While loops are very versatile and useful. However, physicists are often looping over a set of numbers as in the example of evaluating a function for many values. For these applications, we use a `for` loop to loop over all of the elements of a list or array.

First, let's loop over a list of strings. Type this in and try it:

```
cheeseList = ['Wensleydale', 'Stilton', 'Danish Blue', 'Red
Leicester', 'Brie']
for cheese in range(0, len(cheeseList)):
    print "Do you have some", cheeseList[cheese], "?"
    print "Not as such"
```

Note: This could have been written more concisely as:

```
cheeseList = ['Wensleydale', 'Stilton', 'Danish Blue', 'Red
Leicester', 'Brie']
for cheese in cheeseList:
    print "Do you have some", cheese, "?"
    print "Not as such"
```

In the first version, we have used a very useful Python function `range()`. Because we so often want to iterate (or loop) over a list of numbers, Python provides this to create a suitable list of integers.

Try this function out from the command line:

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

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

Note: You can give it a start and end value and an increment, or just an end value. It returns a list up to *one less than* the end parameter.

So now let's use it to loop over a set of numbers:

```
for number in range(10):
     print "The number:", number, "It's square:", number**2
```

Note 1: The variable 'number' is just used to store the element from the list generated by the range function that we use on each go round the loop.

Note 2: As you can see, Python is happy to square the value of 'number' before it prints it.

7.2.3 Getting out of Infinite Loops - break

It is often useful to use an infinite loop, typically using `while True:`. You can break out of such a loop on a condition using the break command. You should not use this more than is necessary as it can make code somewhat less clear to read. Here is an example in 'pseudo code'. This is a useful way of showing some code structure without sticking to the language syntax:

```
while True:
     Do something
     if <some test>:
          break
     else
          do some other stuff
carry on # The break causes a jump to here
```

You can also escape from an infinite loop by entering <CTRL C> from the keyboard.

7.3 Exercises

7.3.1 Exercise 7.1 – *Advanced topic*

Improve the program that you wrote for exercise 6.3 using a dictionary :

- The new program should loop after each name has been dealt with, and exit if the user just hits <return> with no name. It should print a message on exit.
- The program should handle names that are not in the database cleanly by issuing a message and asking for another name

Model solution is in file: Exercise7.1.py on DUO.

7.3.2 Exercise 7.2

Write a program to list some integers, their squares and their cubes. Read the start value, the

end value and the increment from the command line. Try to do this with just one input. Use a `for` loop and the `range()` function. Test your program starting at 2 and ending at 10 with an increment of 2. You should see some input and output like this:

```
Enter Start, Stop, Increment: 2,10,2
Number     Square     Cube
   2        4           8
   4       16          64
   6       36         216
   8       64         512
```

Model solution is in file: Exercise7.2.py on DUO.

7.3.3 Exercise 7.3

Improve the program you wrote for exercise 7.2:

- Make it loop until exit is requested by the user hitting <Return> only.
- Allow the Stop and Increment parameters to be optional. We indicate this using square brackets in the prompt. Supply defaults for when they are not entered.

A session might now look like this:

```
Enter <Carriage return> only to exit
Enter Start, [Stop], [Increment]: 2,6,2
Number     Square     Cube
   2        4           8
   4       16          64
   6       36         216
Enter Start, [Stop], [Increment]: 1,3
Number     Square     Cube
   1        1          1
   2        4          8
   3        9         27
Enter Start, [Stop], [Increment]: 2
Number     Square     Cube
   2        4          8
Enter Start, [Stop], [Increment]:
Exit requested
```

8 Chapter 8 - Functions and Modules

The whole principal of writing software for Physics depends on good program structure. Much of the functionality that you require will be provided by modules that you import into your programs. In a similar way, nearly all of the functionality of your own code should be built as a set of modules (just a separate file), each of which contains a group of related functions.

Every function that you write is a separate entity and can, and should, be tested independently to be sure that it works as you expect it to.

Note 1: You can include one or two functions with your main program in these simple examples, but note that the function definition must occur before the main program. Once all of your functions are in modules, you import them at the top of the main program file.

Note 2: In his wisdom, Hetland calls his chapters on functions *Abstraction;* a very correct name but a touch confusing. His chapter 6 covers the writing and use of *functions*. His chapter 7 covers the alternative technique of using *methods* (a part of object-oriented programming). These are not included in this course but feel free to learn such techniques if you have time. *Beware:* Because the use of methods is becoming more and more popular, many programmers will refer to their functions as methods.

8.1 A First Function

So, let us now look at writing some code as a function and then calling that function from a main program. We want to evaluate the polynomial $x^3 – 7x^2 +14x -5$:

```
def poly(x):
    '''
    Evaluate the polynomial x^3 - 7x^2 + 14x - 5
    '''
    return x**3 - 7*x**2 + 14*x - 5.0


# Test program for the function poly()
while True:
    xValue = float(raw_input("Enter a value for x:"))
    yValue = poly(xValue)
    print "The value of the polynomial is:", yValue
```

There are various points to note here:

Syntax: Note that the line with the `def()` command on it ends in a colon. You must then indent the block of code that is the function, just as you did for conditional blocks. IDLE will help you to do this.

Warning: Be very careful with your indentation. Python relies heavily on indentation to know when a block of code starts and ends. Your indentation must be correct.

You can pass parameters to functions (separated by commas for more than one). The equivalent variable within the function is a different variable. Its value will not affect that in the calling program. In general, use a different name to remind you that they are different. The variable x in the function above is said to be *local* to the function `poly()`.

If you want to return some value to the calling program, you must use the `return` statement.

We could have used another variable for the value of the polynomial in the function and returned that variable but Python is happy not to bother. It works out the value and returns it. This code would be just as valid:

```
y = x**3 - 7*x**2 + 14*x - 5.0
return y
```

A function doesn't have to have a `return` statement. It's often handy to have a few lines of code in a function that do something useful but don't return a result.

This program will run forever! The value of `True` is always True. To escape from the program, just type <CTRL C>, meaning hold down the CTRL key and type C. This will interrupt your program and return you to the Python prompt. It is better to build in a way to exit cleanly as we saw in exercise 7.3.

8.1.1 Default values for parameters in a function

In general, the number of arguments sent to a function should match the number that it is expecting. However, you can pass less than the number it is expecting, provided you give it a default. This is done as follows:

The function definition:

```
def myCleverFunction(start, stop, inc=1):
```

You MUST provide `start` and `stop` but, if you miss out `inc` it will be set to 1.

In this call, the default value of `inc=1` is overridden by the value of 5 provided:

```
start = 1; stop = 10; increment = 5
a = myCleverFunction(start, stop, increment)
```

In this call, no value is provided for `inc`, so it is set to 1:

```
start = 1; stop = 10
a = myCleverFunction(start, stop)
```

8.1.2 Documentation Strings

As well as using a # for comments, you can just put a string on a line on its own as a comment. It is usual to put a *documentation* string at the start of every program, at the start of every function and at the top of every module. These documentation strings are important. Python itself will use them to provide help to the programmer. To see this in action, when you type in a call to your own function using IDLE, it will pop up a 'tip' to say how the function works. This will be your documentation string. We normally use *triple* quotes (of either type) for this. This allows there to be carriage returns within the documentation string.

Note: In your course work, you will be *expected* to provide documentation strings. You will lose marks if they are omitted.

8.2 Returning more than one value

You may wish to return more than one value from a function. This is very easy. The return statement will take a list of values, separated by commas. This should match a set of values in the calling program. Alternatively, the returned values can be put in one variable. This will be a tuple. Just treat it like a list to access the results:

```
def cleverFunction(<Parameters>):
     return a,b
first, second = cleverFunction(<Parameters>)
```

or:

```
def cleverFunction(<Parameters>):
     return a,b
results = cleverFunction(<Parameters>)
a = results[0]
b = results[1]
```

Note: The above is what is known as 'pseudo code'. It is not working Python. It just gives an idea of how some real Python might be written. The non-Python bits are often put inside <>.

8.3 Modules and import

So, what is a module?

It is just a library of useful functions or methods gathered together into a single file. In general, we gather together functions with related functionality into a module. There are very useful standard modules that you can use, and of course you can write your own. Once a module exists, any program can 'import' it and make use of its functions.

Whether it is a standard module or your own module, the ways of importing a module (making it available to your own program) are the same.

Note: A module can import other modules. If you write your own module of clever maths routines, it will probably have to import the `numpy` module in order to have some maths functionality.

Modules are so important they are mentioned in chapter 1 of Hetland. However, the main chapter on modules is his chapter 10 ('Batteries Included'). This gives a neat introduction to using standard modules in Python.

We give just a brief introduction to how to use modules here.

There are several different ways in which modules can be imported:

8.3.1 Import

This is the default way to import a module. If in doubt, always use this plain import. Every call to a function in the module must use the module name as a prefix to the function name, separated by a dot(.):

```
import myCleverLib # import the module
```

```
# Now call a function in that module
theAnswer = myCleverLib.miracle(<parameters>)
```

8.3.2 from <module> import <function>

It is sometimes a bit boring to keep using the full name of a function in a module. There are clear advantages to doing so and it is good practice to do so. The function call, with its module prefix, will be unique which is essential to avoid confusion.

However, as always, there are shortcuts and you will see them used. It is poor practice so try not to use them. If you do, be careful. It's easy to get confused.

You can import one or all of the functions from a module so that they look just like functions in your own module:

```
# Import just the one function from a module
from myCleverLib import miracle
# Now call a function in the module
theAnswer = miracle(<parameters>)


#Import all of the functions from a module
from myCleverLib import *
# Now call two functions from the module
theAnswer = miracle(<parameters>)
theOtherAnswer = theOtherWay(<parameters>)
```

If you have a function with the same name in two different modules and import them like this, you won't know which one you are calling! Even worse, Python will probably decide which one to use and assume that you know. IE It won't give you an error. So beware.

8.3.3 import <module> as <name>

This is just a useful shorthand. If the module has a long name, you can replace that name at import time with a shorter one:

```
#import the module with an alternative name
import myCleverLib as lib
#Now call a function in the module
theAnswer = lib.miracle(<parameters>) # A bit shorter
```

Advanced topic:

Just for fun, there is another shortcut technique. You can use a variable to 'point to' a function:

```
import myCleverLib
clever = myCleverLib.miracle # A pointer to the function
```

```
theAnswer = clever(<parameters>)
```

8.3.4 Allowing Python to find your Modules

Python must know where to find your modules. It keeps a list of where all of the standard modules live so you don't have to worry about them.

The best way to start working is to keep your modules in the same directory from which you are running your main program. This is one of the places where Python will look for modules.

Let's assume you get more organized and put all your well tested modules in the directory: /home/fred/mymodules.

You may be developing a program in a different directory. Python can't find your modules! You need to set what is called an environment variable with the special name PYTHONPATH. This can be done at the Linux prompt with the command:

```
setenv PYTHONPATH /home/fred/mymodules
```

You can check that the variable is set using:

```
echo $PYTHONPATH
```

Now Python will look in that directory as well as all of its usual places.

The value of this environment variable will be lost when you log off from Linux.

This process is described in more detail in Hetland chapter 10. He also describes the different process used if you are working under MS Windows.

Note: If you don't know if a module exists on your system, try to import it from the Python prompt. If it's there and Python can find it, it will import ok.

8.3.5 What's in a Module?

Once you find, or are told, about a useful module, how do you know what is in it? What functions does it offer?

There is a useful built-in function in Python called dir().

This takes the name of a module and tells you what is in it. You will have to import it first of course!

We will soon come across the math module. Try importing it then use dir(math) to find out what it can do:

```
>>> import math
>>> dir(math)
['__doc__', '__name__', 'acos', 'asin', 'atan', 'atan2',
'ceil', 'cos', 'cosh', 'degrees', 'e', 'exp', 'fabs', 'floor',
'fmod', 'frexp', 'hypot', 'ldexp', 'log', 'log10', 'modf',
'pi', 'pow', 'radians', 'sin', 'sinh', 'sqrt', 'tan', 'tanh']
>>> print math.__doc__
This module is always available.  It provides access to the
mathematical functions defined by the C standard.
>>> print math.sin.__doc__
```

```
sin(x)
Return the sine of x (measured in radians).
>>>
```

Right away, we can see a list of the functions in the module. We have printed the 'attribute' called `__doc__`. This gives you the documentation string or comment that the programmer put (as you should put) at the head of his module.

Every function in the module should also have such a documentation attribute. We have printed the one for the `math.sin()` function.

Python built-in functions also have __doc__ attributes. Try typing this at the command line:

```
print range.__doc__
```

Note 1: Python uses some special names within modules and functions that begin and end with two underscore characters. Don't use such names for your own variables!

Note 2: IDLE will help you when you are typing, either in the Python shell or in the editor, by putting up 'tips', or little pop-up boxes to remind you of the relevant syntax. These are based on attributes of the function or module such as `__doc__`.

8.3.6 Testing Functions and Modules

Testing every function and every module that you write is essential. When you write a new function, test it with some test code then put it into a relevant module. A module file contains one or more functions and is designed to be imported, not run on its own. However, it is useful to the programmer who tests and maintains the module to be able to test the functions by just running the module. There is a standard way to do this. The module will have a (hidden) variable called `__name__`. If you just run a module, this name will be set to the default which is "`__main__`". You can thus detect that the user is running the module, rather than importing it, and run a bit of test code. This bit of code should be at the end of the module. You can change it to cause a call to the function being tested, and to print some results as in this example:

```
''' Module to do some fancy stuff '''
def clever(x,y):
     ' Function that is pretty smart'
     < Do clever stuff > # The clever code goes here
     return result

if __name__ == '__main__':
     print 'Testing clever()'
     a = 1.2
     b = 3.4
     answer = clever(a,b)
     print 'a=%.3f, b=%.3f, answer = %.3f' % (a,b,answer)
```

This is a standard way of testing a function in a module. It is also fine of course to write a dedicated test program that imports the module and calls the function. If you leave test code in a module that is not subject to this test on `__name__`, it will be executed on import to your main program; probably NOT what you want.

8.3.7 pyc Files

When you import a module, in order to speed up execution, Python will 'compile' the module and put the compiled code in a file with the same name but a .pyc extension. 'But this is an interpreted language isn't it?' Well yes. This is not a real compilation but a conversion to something called 'byte code'. Next time you import the module, Python will use this compiled version. If you change the source code of the compiled module, Python will notice and recompile it next time you use it.

The point is that the byte code in the .pyc module will run much faster than if the module is interpreted every time.

You don't have to worry about any of this. Just ignore the .pyc files. If you delete them, Python will re-create them when it needs to

8.4 Exercises

8.4.1 Exercise 8.1

Write a program to evaluate the polynomial $x^3 - 7x^2 + 14x - 5$ as described in section 8.1. You can improve on the code given in that section! For this first function example, just put the function in a file followed by some code to test it.

Model solution is in file: exercise8.1.py on DUO.

8.4.2 Exercise 8.2

Improve the program you wrote for exercise 8.1 by moving the function into a separate file; a new module. Put a documentation string in the module and in the function. Add some standard test code as described in section 8.3.6. Then write a separate test program that imports your new module and uses the function. You can use much the same code for testing within the module and in the stand-alone test program.

Model solution for the module is in file: poly01.py on DUO.

Model solution for the test code is in file: exercise8.2.py on DUO.

8.4.3 Exercise 8.3

Write a function that will evaluate ANY cubic (order 3) polynomial. Add this function to your module. It should take both the value at which to evaluate the polynomial and the 4 coefficients as parameters. Hint: The coefficients could be passed as a list. If the coefficients are not passed to the function, they should take on default values. Write some simple code to test this function inside the module.

Model solution for the module is in file: poly02.py on DUO.

Now write a program that uses the function. Ask the user to input the coefficients. Enumerate the function from -5 to 5 in steps of 0.5 and print the results to the screen. Test the function for various sets of coefficients.

A model test program is in the file exercise8.3.py on DUO.

8.4.4 Exercise 8.4

Write a function that takes a list of real numbers and returns both the maximum and the minimum values. Put the function in a module file with some simple test code. Make it work for positive and negative floating point numbers.

Model solution for the module is in file: myNumericLib01.py on DUO.

Now write a test program for this function. Rather than ask for the numbers from the screen, generate a set of test values in a list. Print the test data and the maximum and minimum to the screen in the test program. Check that the function works as expected.

A model test program is in the file exercise8.4.py on DUO.

Note: There are of course built-in functions called max() and min() in Python. They are useful but don't call your own variables 'max' or 'min'. Python may not do what you expect! You should not use variable or function names that already exist in Python. If you don't know if Python already has a function or keyword, try it at the command line.

8.4.5 Exercise 8.5

Add two further functions to the module you developed in exercise 8.4 to find the mean and the sum of a list of numbers.

Hint: Use your sum function in your mean function.

Beware: Python has a built-in function called `sum()`.

Model solution for the module is in file: myNumericLib02.py on DUO.

Modify the test program of exercise 8.4 to use all three of the functions in your new module.

A model test program is in the file exercise8.5.py on DUO.

9 Chapter 9 - Debugging and Exceptions

At some point, your program will go wrong! This may be an error that crashed the program and gives you a traceback (the easy ones to find), or it may be an error that means that the program runs to completion but gives you the wrong answer (harder to find). There are various ways to find out where the bug is.

We tend to call all errors in programs 'bugs' (for historical reasons). In Python, errors are properly referred to as 'exceptions'. Dealing with exceptions is an important part of programming. Chapter 8 in Hetland covers this. We give a brief introduction here.

The handling of exceptions is an *advanced topic* so you can skip most of this section if you wish. Just note the main debugging methods mentioned in sections 9.1, 9.2 and 9.3.

9.1 Using `print` for debugging

The first and foremost method of debugging is to print out any variables that you are using that might have the wrong values. Just add a print statement to the program, re-run it and check that the variables all have the value that you are expecting:

```
print 'a=', a, ' b=', b # This is a debug line
```

9.2 Use the Command Line

If you get an error that suggests you have used the wrong syntax or called a function incorrectly, repeat what the program was doing when it crashed at the command line. All Python programmers, however skilled, do this. Get it to work at the command prompt then use it in your program.

9.3 Module Test Code

You can test and debug the functions in your module using test code included in the module.

This technique was described in section 8.3.6

9.4 Handling Exceptions: try / except – *Advanced Topic*

You should, of course, try to remove all of the bugs from your program before it is used to solve a problem. However, there are situations where an error could still occur and you don't want the program to crash when it does.

To prevent this happening, there is a construct called `try / except` that tells Python what to do when the error occurs.

You don't need to catch exceptions in this course, just fix every bug!! However, exception handling is a powerful advanced technique.

9.4.1 Catching ALL Exceptions

The handling of exceptions is covered in chapter 8 of Hetland. There is not time in this course to go into exception handling in any detail. However, let's just look at one simple example because the `try / except` construct can be very useful. Try this program:

```
a = input('Enter a number: ')
b = input('Enter another number: ')
print a/b
```

If you enter 0 for the second number, you should get this output:

```
Enter a number: 2
Enter another number: 0
Traceback (most recent call last):
  File "testException.py", line 3, in <module>
    print a/b
ZeroDivisionError: integer division or modulo by zero
```

You can stop the program from crashing by 'catching' the exception:

```
try:
    a = input('Enter a number: ')
    b = input('Enter another number: ')
    print a/b
except: # Catch ALL exceptions
    print 'The second number must not be zero'
```

Run this as before and you should get:

```
Enter a number: 2
Enter another number: 0
The second number must not be zero
```

Here, Python has seen the division by zero, but it has also seen your 'exception handler'. This is the indented code after the `except` which it has duly executed. The program did not crash! If there is no exception, the exception handling code is just ignored.

Note 1: Take note of the colon used after try and except and the fact that, like a conditional block, the code within the `try` or `except` block must be indented.

9.4.2 Catching Specific Exceptions

The above example will catch ALL exceptions that occur. This is generally bad practice. If there is a syntax error in your 'try' block, the exception handler will catch it and give you no traceback!! You should specify the exceptions that are likely to occur.

It is possible to catch only one (or more) specific exceptions. This is well described in Hetland, chapter 8.

Here is some pseudo code as an example:

```
try:
     <Some smart code>
except (<exception1>, <exception2>):
```

```
    print "An exception occurred"
```

Only the `try` block is normally executed. If either of the specified exceptions occurs, the `except` block is executed. If any non-specified exception occurs, the program will crash.

Note 1: If there is more than one specified exception, put them in a tuple, as shown.

Note 2: To find out the name of an exception, make it happen at the command line and look in the traceback:

```
>>> a=2/0
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    a=2/0
ZeroDivisionError: integer division or modulo by zero
```

The name of the exception here is 'ZeroDivisionError'.

The code from section 9.4.1 should therefore have been written as below. Note that we have put only the code that might be subject to the error inside the `try` block:

```
a = input('Enter a number: ')
b = input('Enter another number: ')
try:
    print a/b
except ZeroDivisionError: # Catch division by zero
    print 'The second number must not be zero'
```

9.5 Exercises

9.5.1 Exercise 9.1 – *Advanced Topic*

Write a routine that requests input of several floating point values from the screen, catches errors on input and returns the values. Put the function in a module file with some simple test code. The function should take a prompt string as its single parameter. It should return all of the valid numbers that are input as a list of floats. If there is some invalid input, it should return an empty list.

Test the function with some valid input and some invalid input. The two tests should produce output like:

```
Enter the parameters separated by commas:1,2,3
The parameters are: [1.0, 2.0, 3.0]
>>>
Enter the parameters separated by commas:1,2,3,fred
The parameters are: []
```

Hint: Use the `try/except` construct to catch errors where they are likely to occur.

A model solution is in the file exercise9.1.py on DUO.

10 Chapter 10 - Maths Modules: NumPy

As yet, we have not really done much mathematics! The Python language does not include built-in support for maths. However, there are of course modules that we can import to provide mathematics functions. The most basic is the math module. This is generally not enough for the Physicist. By all means load the math module and try it. However, we will use the far more powerful NumPy module.

This was introduced briefly in section 6.3 but further details are given here.

10.1 The math Module

This provides basic maths functionality: sqrt(), sin(), cos() etc.

10.2 The NumPy Module

A huge amount of development has gone into the development of a mathematics module called NumPy.

Note 1: It is usually written as NumPy but when you import it use: `import numpy` (all lowercase).

Note 2: You may come across previous versions called Numeric and NumArray. These are now out of date. Please don't use them.

The `NumPy` module has been installed for you on the ITS Linux service. It is also bundled with the *Enthought* version of Python that we recommend for MS Windows.

Hetland does not cover the NumPy module at all. However, as stated previously, there is an excellent web page that provides an introduction and a tutorial at: http://www.scipy.org/.

This web page refers to another module, SciPy, which provides more complex mathematical functionality, but is heavily dependent on NumPy. Learn to use the facilities of NumPy first. We will give a very brief introduction here. It is worth trying the tutorial which can be found at: http://www.scipy.org/Tentative_NumPy_Tutorial.

What NumPy gives you is new numerical array and matrix types and a set of basic operations on them. Unlike Python lists, this new *array* type is a set of elements, *all of the same type*. Thus these arrays can be used for vectors and matrices. They provide us with fast matrix and vector manipulation.

One thing to note is that these new types are implemented in pre-compiled code in the C language. The important consequence is that, once you define one, it has some memory allocated to it. You cannot therefore increase its overall size once it has been created.

So, as always, let's try some things at the command line:

10.2.1 Creating Arrays and Some Examples of Basic Manipulation

There are various ways to create an array:

You can create an array from a list:

```
>>> import numpy
>>> a = numpy.array([1.0,2.0,3.0])
>>> type(a)
```

```
<type 'numpy.ndarray'>
>>> a
array([ 1.,  2.,  3.])
>>>
```

Note 1: The type of an array is actually *ndarray* for 'n-dimensional array'.

Note 2: We can have matrices of any number of dimensions.

You can create an array using the NumPy equivalent of `range()` which is called `arange()`:

```
>>> y = numpy.arange(12)
>>> y
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11])
>>> y.shape = 3,4 # Make it a 3 by 4 matrix
>>> y
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
>>>
```

We have created y then changed its shape attribute. In maths terms, we started with a 1d vector and changed it into a 3x4 matrix. You can *only* do this if the total number of elements remains unchanged.

Now we can do some maths on our matrix:

```
>>> a = y*3
>>> a
array([[ 0,  3,  6,  9],
       [12, 15, 18, 21],
       [24, 27, 30, 33]])
```

Without the matrix capabilities of NumPy, you would have to do this multiply one element at a time in two nested `for` loops. Try it! It's a useful exercise to try once.

You can slice an array and index an array just as you would a Python list. Use commas to separate the dimensions and the usual colon for the slice:

```
>>> a[2] # The third (numbered from 0!) row
array([24, 27, 30, 33])
>>> a[1,2:4] # The third and fourth elements of the second row
array([18, 21])
```

This can be confusing (mainly because of starting at zero and giving the end index as the one after the last one you want). Try lots of examples at the command line until you get the hang

of it.

We can change the value of any element or slice of the matrix:

```
>>> a[2,3] = -99 # Change the fourth element of the third row
>>> a
array([[  0,   3,   6,   9],
       [ 12,  15,  18,  21],
       [ 24,  27,  30, -99]])
>>> a[1] = 0 # Set the second row to all zeros
>>> a
array([[  0,   3,   6,   9],
       [  0,   0,   0,   0],
       [ 24,  27,  30, -99]])
```

Our array has various useful attributes that we can access:

```
>>> a.shape # The shape of a
(3, 4)
>>> a.ndim # The number of dimensions of a
2
>>> a.size # The overall size of a
12
```

So far, we used integers; but `arange()`, unlike `range()`, can take floats as parameters:

```
>>> numpy.arange(2.0,6.0,0.5)
array([ 2. ,  2.5,  3. ,  3.5,  4. ,  4.5,  5. ,  5.5])
```

There is a function to create an array with all zeros. This can then be filled with the data we want:

```
>>> a = numpy.zeros(10)
>>> a
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0])
>>> type(a[0])
<type 'int32scalar'>
```

We didn't tell Python the type of numbers we wanted in the array. It has given us integers by default. Some versions of NumPy will give you floats by default. To be sure you get what you want, you can specify the type using `dtype` as follows:

```
>>> a = numpy.zeros(10,dtype=numpy.float64)
>>> a
array([ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.])
>>> type(a[0])
<type 'float64scalar'>
```

There is also a function to fill the new array with ones:

```
>>> numpy.ones(10, dtype=numpy.float64)
array([ 1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.])
```

10.2.2 Linear Algebra

NumPy provides a LOT of useful matrix and vector manipulation routines; too many to list here. Let's just look at one basic linear algebra routine that you might need:

If you use the normal * for multiplication, NumPy will multiply all of the elements of two matrices together:

```
>>> a = numpy.arange(1,5,dtype=numpy.float)
>>> a
array([ 1.,  2.,  3.,  4.])
>>> a.shape = 2,2
>>> a
array([[ 1.,  2.],
       [ 3.,  4.]])
>>> b = numpy.arange(1,5,dtype=numpy.float64)
>>> b
array([ 1.,  2.,  3.,  4.])
>>> b.shape = 2,2
>>> c=a*b
>>> c
array([[  1.,   4.],
       [  9.,  16.]])
```

This may not be what you want. If you want the dot product, use the function `numpy.dot()`:

```
>>> d = numpy.dot(a,b)
>>> d
array([[  7.,  10.],
       [ 15.,  22.]])
```

Check this is correct by hand.

Many of the more advanced linear algebra functions are contained in a sub-module within NumPy called `numpy.linalg`. Have a look at what is available using:

```
>>> import numpy
>>> dir(numpy.linalg)
['LinAlgError', '__builtins__', '__doc__', '__file__',
'__name__', '__path__', 'cholesky', 'det', 'eig', 'eigh',
'eigvals', 'eigvalsh', 'info', 'inv', 'lapack_lite', 'linalg',
'lstsq', 'norm', 'pinv', 'qr', 'solve', 'svd', 'tensorinv',
'tensorsolve', 'test']
```

10.3 The SciPy Module – *Advanced Topic*

The NumPy module contains a large number of useful functions for numerical calculations. The SciPy module extends these facilities. Many of the functions in SciPy are higher level routines that build on the facilities provided by NumPy.

You don't need to use SciPy in this course but you will find it useful in your future courses on Computational Physics. There are documentation and examples on-line at:

www.scipy.org

10.4 Exercises

10.4.1 Exercise 10.1

Write your own routine to find the dot product of two matrices *without* using `numpy.dot()`. This can be done using three nested loops. Test it with two off 2x2 matrices and check your result by hand. Test it with 2 off 3x3 matrices and check your result using numpy.dot().

A model solution is in the file exercise10.1.py on DUO.

10.4.2 Exercise 10.2

This is an example of some code that is practical and useful. It is a little harder than previous examples.

When we take an astronomical image, the pupil of the telescope is circular due to the circular primary mirror. However, the CCD detector is square. When we come to analyse the CCD data, we don't want to bother with the pixels that are outside the pupil. We therefore need a mask that has a '1' for every pixel inside the pupil, and a '0' outside it.

Write a program that will make such a mask for any square array of data and any pupil radius. There are many ways of doing this. One way is to generate an array that, for every element or pixel, contains the distance of that pixel from the centre of the array. The mask elements are then just '1' when this value is less than the pupil radius and '0' if it is greater.

Hints: There is a useful NumPy method called `numpy.where()`. This sets elements of an array to one value or another depending on a condition.

The 2-d array of distances from the centre can be filled by using two nested loops. This is rather slow. There are various faster ways of doing this without loops using NumPy techniques. To map values from a 1-d array onto a 2-d array, the 1-d array must be extended to 2-d. There is a NumPy facility called 'newaxis' that will add a dummy dimension to an

array.

A model solution is in the file exercise10.2.py on DUO. This module contains two functions. One is based on using loops and the other, faster function uses NumPy facilities.

11 Chapter 11 - File Input and Output – The Details

So far, we have looked mostly at input and output to the screen. You will often want to do some calculation and output the results to a file. Similarly, you may need to read some data from a file in order to analyse it. You don't have to worry about how the computer stores its data on the hard disk. The operating system will look after that. All you have to do is tell Python which file you want to use, and how you want to use it.

We gave a short introduction to doing this in section 4.2 that allowed you to read and write NumPy arrays to/from file. However, you may want to output quite complex mixtures of text and numbers to a file. The details of file IO are given here and there is a self-contained short chapter (11) in Hetland on 'Files and Stuff'.

11.1 Line Terminators – The \n character

It is often useful to read individual *lines* of text from a file. The end of a line is indicated by a special character or characters. The characters used vary from language to language and for different operating systems. Python uses the character \n (backslash n). This represents a single End of Line (EOL) character.

Try this at the command line:

```
>>> a='This is the first line. This is the second'
>>> print a
This is the first line. This is the second
>>> a='This is the first line.\nThis is the second.'
>>> print a
This is the first line.
This is the second.
```

As you can see, the print function translates the \n into a new line on the screen.

Python is very good at dealing with new line characters. Unix also uses \n so no problem there. MS Windows does not. Python on MS windows will do all the translating for you so you don't have to worry. Just use \n when you want a new line.

When writing text to a file, separate the lines with a \n. You can then use the readline() and readlines() functions when you read the file.

11.2 Writing to File

Let's first write some text to a file, then we can try reading it back in various ways. First of all, you must open a file. Not surprisingly, we open a file using the `open()` function. It will return a file object. We use this object and its methods to access the file. We are again slipping into using objects and methods rather than functions. That is how Python works and you know the syntax so just use the methods provided.

Note 1: Before you write to a file, remember that the file will be written in the directory in which you are working unless you give a full path name. To find the name of your working directory under Linux, type pwd (print working directory).

To write to a file, try this at the command line:

```
>>> f = open('temp.txt', 'w')
>>> f
<open file 'temp.txt', mode 'w' at 0x0138D260>
```

The variable f is a file pointer object.

The second parameter of the open function tells Python how to use the file:

'r' => You may only read from this file. This is usually the default.

'w' => You may only write to the file

'a' => You may append to the file. IE You will write *after* any data already in the file.

Note 2: It is also possible to read and write to a file using 'r+' or 'w+' mode in combination with the `seek()` method to read or write to any point within the data in an existing file. You probably won't need this yet so it is not described here.

Now write some lines to the file:

```
>>> f.write('This is the first line.\n')
>>> f.write('This is the second line.\n')
>>> f.write('This is yet another boring line.\n')
>>> f.close()
```

Notice that we close the file when we have finished with it. You should not leave files open.

You will find that you now have a file called temp.txt in your working directory. (Check under Linux with the `ls` command). Try typing the file – under Linux type: `more temp.txt`.

Note 3: After each write to the file, we have put in a \n to split the file into lines.

Note 4: By default, Python will always write from the beginning of a file that is opened in 'w' mode. So *beware* of overwriting any existing contents.

Note 5: All of your data should be written to file as strings. Thus you must convert any numbers to strings before writing them to file. You can write numbers to file in 'binary' mode but this is not needed for this course.

11.3 Reading from File

Reading from a file is equally easy. You must however tell Python how much of the file you want to read. In general you can:

Read a fixed number of bytes from a file using the `read(num)` method where `num` is the number of bytes to read. This is not very useful!

Read a single line from a file using the `readline()` method.

Read all the lines in a file using the `readlines()` method. They will be returned as a Python list.

Note: After a read, Python keeps a pointer to where you got to in the file. If you do another read, it will start from there.

Try reading from the file that you created from the command line:

```
>>> f = open('temp.txt')
>>> f.readline()
'This is the first line.\n'
>>> f.readlines()
['This is the second line.\n', 'This is yet another boring
line.\n']
```

11.4 Exercises

The first two exercises expect you to output some mixed data to a file. We output the number of records as an integer 'header' to the file. These exercises are intended to give you some practice at detailed file IO. Do the first two exercises *without* using the NumPy savetxt() and loadtxt() functions. Then repeat the exercises using the NumPy functions. Refer back to section 4.2 to remind you how to do this.

11.4.1 Exercise 11.1

Write a program to calculate the sine of some angles from 0 to 100 degrees. (Remember to convert to radians before taking the sine). Use numpy arrays for this. Output the data to the screen so that you can see it as it is generated. Output the data to file. On the first line of the file, output the number of records that you are writing to file (100). This acts as a useful 'header' so that you can easily find out how much data is in the file. Output the data to file with one angle in degrees and its sin on each line. Separate the data with a comma. You can use any separator, but a comma is usual. You are creating a 'comma separated variable (CSV) file. Look at the file you have produced and check it is correct.

Model solution is in file exercise11.1.py on DUO.

11.4.2 Exercise 11.2

Write a program to read your angle and sin data from the file into numpy arrays and write every 10^{th} angle and its sine to the screen with one angle and its sine on each line in the format:

```
angle =  0.000 degrees      Sin = 0.000
angle = 10.000 degrees      Sin = 0.174
```

etc

Read the 'header' to find out how much data is in the file and use this to size your arrays.

Try to get the output data to line up as shown using the % format descriptor.

Model solution is in file exercise11.2.py on DUO.

11.4.3 Exercise 11.3

Repeat exercise 11.1 using the NumPy savetxt() function but do not write the number of records as a header since this is not needed.

Model solution is in file exercise11.3.py on DUO.

11.4.4 Exercise 11.4

Repeat exercise 11.2 using the NumPy loadtxt() function.

Model solution is in file exercise11.4.py on DUO.

12 Chapter 12 - Plotting Graphs

Python does not have any facilities built in for drawing graphs. However, everyone wants to draw graphs so there are many modules around that do so. The matplotlib package is one of the most common and has been provided for you on the ITS Linux service. It goes by the name of PyLab so, if you want a graph, just `import pylab`. The module provides a nice Graphical User Interface (GUI) for you to use to display your graph. Pylab just provides some higher level facilities such as the GUI. It imports matplotlib so just import pylab and you will get both.

There is a good introduction to matplotlib on the web at:

http://matplotlib.sourceforge.net/

Look at the 'Gallery' to see examples of what can be done using this package.

As with all things Python, it is easy to draw a simple graph very quickly. The module provides all the extra facilities that you might need such as labels for axes, different coloured symbols, different types of plot etc. You can slowly learn these and use them to produce good looking clear displays of your data. Just a few of the basics will be enough for this course. It is worth gaining some experience with this package as you will be expected to use it to draw graphs in later courses in level 2 and in level 3.

We do not explain any of the detail of PyLab here. You should get that from the documentation on the web. We include some general points of interest and an example for you to try. Plotting graphs is great fun but don't spend too much time on it!

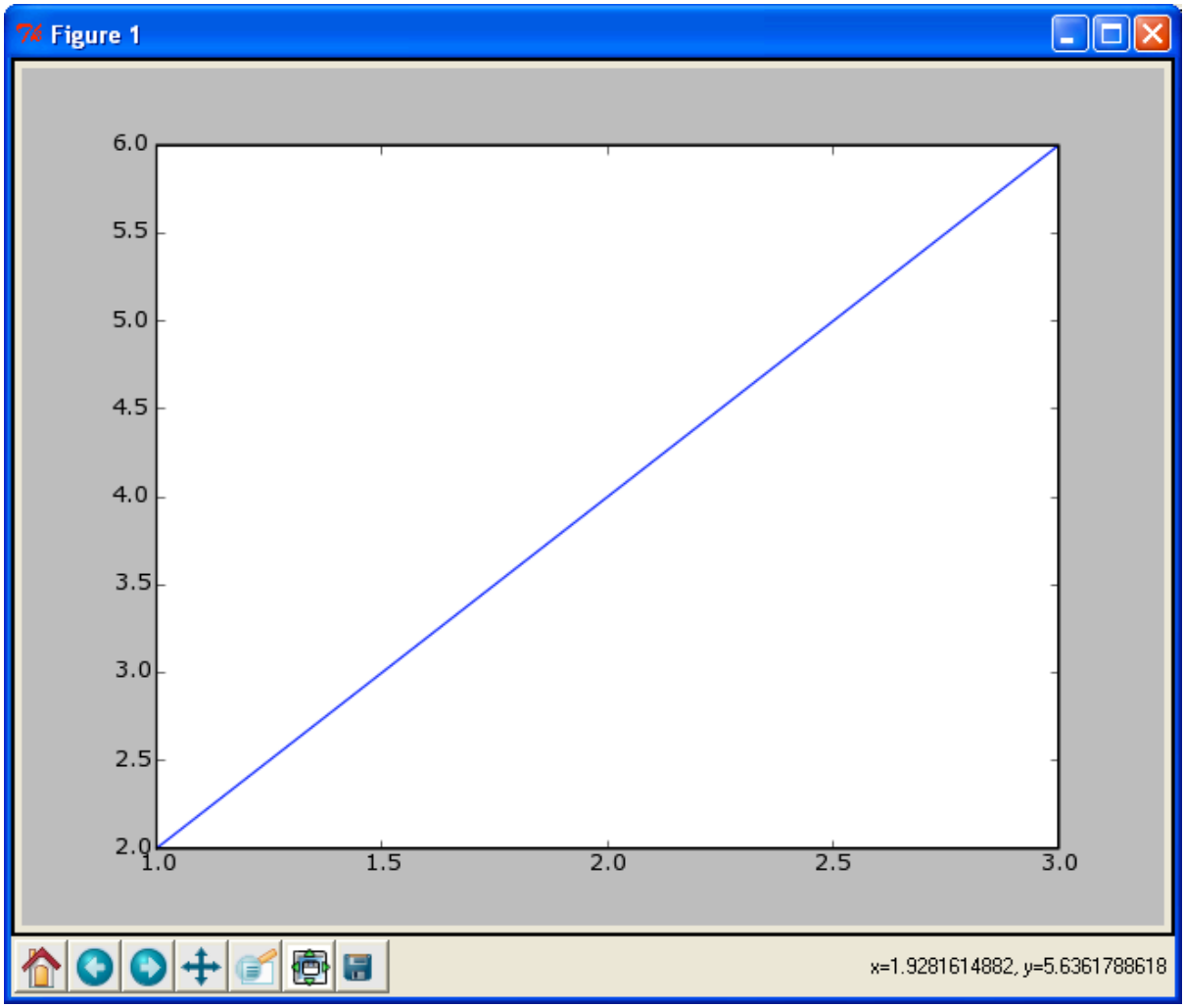
12.1 PyLab: The absolute Basics

What do we always do with something new in Python? Try it at the command line. However, as we state in the next section, things like plotters often get confused if you run them in the IDLE environment. So do this at a Python prompt in your terminal window. (IE type python <myprog> at the Linux prompt).

The basic plot command is `plot(a,b)` where a and b are lists (or arrays) of data to plot in x and y respectively.

```
>>> import pylab
>>> a=[1,2,3]
>>> b=[2,4,6]
>>> pylab.plot(a,b)
[<matplotlib.lines.Line2D instance at 0x01A03170>]
>>> pylab.show()
```

You should now see a plot popup something like this:



This is an example of a Graphical User Interface or GUI. It has been written for you. You are just using it from your program. PyLab has plotted your two lists of data against each other. Kill the GUI window and you should get your Python prompt back.

Note: PyLab has been set up for you to work from a file rather than *interactively* from the command line so you should develop any real programs in a script file as usual.

12.2 GUIs – How do they work?

The last thing you do in your program, if you are going to plot a graph, is to issue the command `pylab.show()`. This causes pylab to draw the Graphical User Interface (GUI) that is used to display your graph. It then enters an infinite loop waiting for you to use the controls on the GUI. Thus you won't get your Python prompt back until you kill the GUI window. If you have problems, you can usually cause the program to end with a Control C (Hold down control and hit c). If you can't get the prompt back in the IDLE Python shell, you will have to close it to stop the program running. As already stated, it is often better not to run such programs inside IDLE. It can get confused!

On Linux, you can run your program 'in the background'. This is the usual way to run GUIs on Linux. Just add the & character to the end of the line when you run your program:

```
python myprog.py &
```

If you now hit the return key in your terminal window, you will get your prompt back. You

can go on working with the GUI still up and running. You *can't* do this in IDLE!

Note: Writing your own GUIs (like PyLab) is great fun but outside the remit of this course. Chapter 12 of Hetland is a good introduction. To write a GUI requires an extra module called a 'GUI builder' which we have not installed for you on ITS Linux. *wxPython* is the current favourite. You can download it to your own computer if you want to try it. Warning: It can be addictive and time consuming!

12.3 Exercises

12.3.1 Exercise 12.1

Modify your program that writes a file of angles and their sines to cover the range 0 to 360 degrees. Modify your program that reads the file to plot this full cycle of the sine function using PyLab. You should give the axes legends and add a title to the graph. Output the graph to a .png file. This could then be added as a figure in a report written in Word or another word processor.

Model solution is in file exercise12.1.py on DUO.

13 Chapter 13 - Random Numbers

There are many problems in Physics where the Physics is well understood but describing an entire system that we wish to model is not possible analytically. In this situation, the Physicist will often resort to a method known as 'Monte Carlo'. This involves using random numbers to repeatedly test what happens to, for example, a cosmic ray interacting with the Earth's atmosphere. (It's like throwing the dice in a casino, hence Monte Carlo).

Python (like all languages) incorporates a "random" number generator. Of course such numbers cannot be truly random. Python can however generate a "pseudo-random" sequence of numbers. We will not address the use of random numbers in any detail here, but you should be aware that there is a module, called 'random', that provides random numbers in a wide variety of different ways.

Import the random module and use dir(random) to see what calls are available. The module is described in detail in the Python documentation in the Module Index at:

http://docs.python.org/modindex.html

The most basic call is `random.random()`. This will return a random floating point number between 0.0 and 1.0.

However, the most useful is probably `random.uniform(<min>, <max>)` which returns a random floating point number between min and max.

The simplest distribution of random numbers you might want would be a 'flat' distribution. One could generate a large number of random numbers using say `uniform(0.0, 100.0)`, bin them in some way and do some tests to see if they are truly random.

One of the other most useful distributions of random numbers is a Gaussian distribution. The module `random` provides the function `random.gauss( <mu>, <sigma>)` to give you just such a distribution where `mu` is the mean and `sigma` is the standard deviation.

13.1 Exercises

13.1.1 Exercise 13.1

Write a program to request a number of Gaussian distributed random numbers. Put them in an array and make a histogram of the array. Use a mean of 100.0 and a standard deviation of 15.0. You can normalise the output so that the integral is 1.0 and the Y axis will represent probability. This also makes the y axis scaling easier for different numbers of random numbers.

Hint: There is a pylab.hist() function to plot histograms.

Model solution is in file exercise13.1.py on DUO.