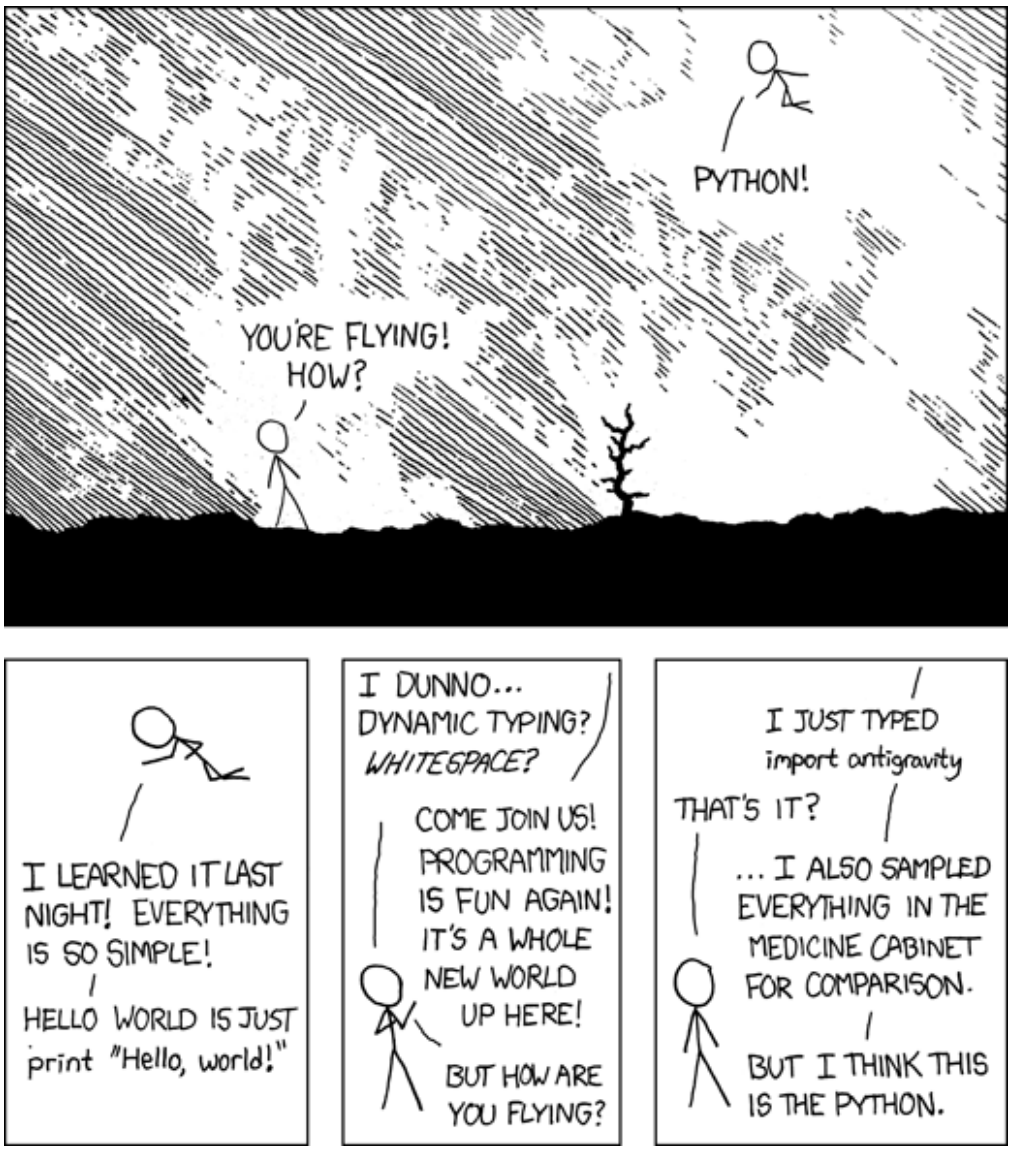
PIP Python Lab Book

University of Sussex

W. Roper; R. Wilkinson; D. J. Turner; M. Hubbard; A.K Romer; J. Loveday
Course Convener: Dr Jon Loveday
Last Update: September 25, 2023



Contents

1 Introduction **v**

2 Some Useful Terminology **viii**

3 Jupyter: Quick reference **1**

4 The Basics **2**
- 4.1 Learning Outcomes . . . 2
- 4.2 Preamble . . . 2
- 4.3 Running `Python` on the University computers . . . 3
  - 4.3.1 Running `Anaconda` for the first time . . . 3
  - 4.3.2 Running `Anaconda` in subsequent sessions . . . 4
  - 4.3.3 Running Jupyter Notebook . . . 4
- 4.4 Running `Python` on your own computer . . . 4
- 4.5 "Hello World" . . . 8
  - 4.5.1 Saving and logging your work . . . 8
  - 4.5.2 Infinite loops happen... and are very bad . . . 9
- 4.6 Data types in `Python` . . . 9
- 4.7 Simple calculations with `Python` . . . 10
- 4.8 Using variables for simple maths operations . . . 13
  - 4.8.1 Test your understanding . . . 14
- 4.9 Changing A Variable's Data Type . . . 14
- 4.10 Commenting . . . 16
- 4.11 Testing the type of a variable . . . 16
- 4.12 A note on the horrors of the insert key . . . 17
- 4.13 Integrated Development Environments . . . 18

5 Strings, Booleans, Tuples and Lists **20**
- 5.1 Learning Outcomes . . . 20
- 5.2 Preamble . . . 20
  - 5.2.1 A reminder about collusion and plagiarism . . . 20
  - 5.2.2 Sourcing Help . . . 20
- 5.3 String manipulations . . . 21
  - 5.3.1 Exercises . . . 22
- 5.4 Boolean Variables . . . 23

5.4.1 Exercises . . . 24
5.5 Tuples & Lists . . . 24
5.5.1 Tuples . . . 25
5.5.2 Lists . . . 27
5.5.3 The `is` Keyword . . . 29
5.5.4 Exercises . . . 30
5.6 Input function . . . 31
5.6.1 Issues with `jupyter` . . . 31
5.6.2 Exercises . . . 32
5.7 Signpost . . . 32
5.8 Advanced Exercises - Some independent learning may be required . . . 33
5.9 Worked Solutions for Section 5.3.1 . . . 34

6 Conditionals, Loops and Functions 35
6.1 Learning Outcomes . . . 35
6.2 Colons and Indentation . . . 35
6.3 The if statement . . . 37
6.3.1 Exercises . . . 41
6.4 While Loops . . . 42
6.4.1 Exercises . . . 44
6.5 For Loops . . . 45
6.6 Format strings . . . 46
6.7 Range command . . . 47
6.7.1 Using the separator command . . . 48
6.7.2 Exercises . . . 48
6.8 Functions . . . 49
6.8.1 Exercises . . . 51
6.9 Signpost . . . 52
6.10 Lambda statement . . . 52
6.11 Dictionaries . . . 52
6.11.1 A Note On Iterators . . . 56
6.12 Advanced Exercises . . . 56
6.13 Worked Examples . . . 57

7 Python Modules and an Introduction to Plotting 59
7.1 Learning Outcomes . . . 59
7.2 What are modules? . . . 59
7.3 Common Python modules . . . 60
7.4 Importing Modules . . . 61
7.4.1 Method One . . . 62
7.4.2 Method Two . . . 62
7.4.3 Method Three . . . 62
7.4.4 Method Four . . . 63
7.5 Accessing help with modules and functions . . . 64
7.5.1 The `dir` command . . . 64

7.5.2 The help command . . . 64
7.5.3 Exercises . . . 65
7.6 The random module . . . 66
7.6.1 Exercises . . . 66
7.6.2 Seeding . . . 67
7.7 The arange command . . . 67
7.8 The matplotlib.pyplot.plot function . . . 68
7.8.1 Exercises . . . 71
7.9 Plotting in multiple figures . . . 72
7.10 Plotting a Scatter with plt.scatter . . . 73
7.10.1 Exercises . . . 74
7.11 Signpost . . . 74
7.12 Advanced Exercises . . . 75
7.13 Worked Examples . . . 75

8 Introduction to Data and Pandas 77
8.1 Learning Outcomes . . . 77
8.2 Loading CSV files . . . 77
8.2.1 Specifying Directories . . . 78
8.2.2 The csv Module . . . 78
8.2.3 The pandas Module . . . 81
8.2.4 Exercises . . . 85
8.3 Plotting data points and error bars . . . 86
8.3.1 Data points . . . 86
8.3.2 Plotting error bars . . . 87
8.4 Fitting straight lines to data points . . . 91
8.4.1 Exercises . . . 92
8.5 Answers to Exercises from Sec 8.3.2 . . . 93

9 Numpy and Arrays 95
9.1 Learning Outcomes . . . 95
9.2 What is numpy? . . . 95
9.3 Array creation . . . 97
9.4 Computations with Arrays . . . 99
9.4.1 Exercises (with worked solutions) . . . 101
9.4.2 Broadcasting . . . 101
9.4.3 Extraction and Shape/Dimension Manipulations . . . 102
9.5 Using numpy's Random Functions . . . 104
9.5.1 Exercises . . . 104
9.6 Treating Arrays as Matrices . . . 105
9.7 The Where Function . . . 106
9.7.1 Exercises (with worked solutions) . . . 107
9.7.2 Advanced Exercises . . . 107
9.8 Optimisation with numpy . . . 108
9.9 Worked Solutions . . . 109

10 Scientific Python **111**
10.1 Learning Outcomes . . . 111
10.2 Opening Different File Types . . . 111
10.2.1 `with` and the importance of closing files . . . 112
10.2.2 Text and binary files . . . 113
10.2.3 Pickle files . . . 113
10.2.4 `Numpy` files . . . 114
10.2.5 FITS files . . . 114
10.2.6 HDF5 files . . . 115
10.3 Treating Images As Arrays . . . 116
10.3.1 Producing images . . . 116
10.3.2 Manipulating images . . . 119
10.3.3 Exercises (with worked solutions) . . . 122
10.4 Histograms . . . 123
10.5 Basic Curve Fitting . . . 126
10.6 Integration . . . 128
10.6.1 Exercises (with worked solutions) . . . 129
10.7 Worked Solutions . . . 130

Chapter 1
Introduction

Preamble

The `Python` programming language is ubiquitous in science, especially in Physics. It is an extremely powerful tool which can be utilised for everything from making simple games to simulating the evolution of the Universe. In fact, 'Big Tech' companies such as Google and Facebook use `Python` behind the scenes of many services you use daily.

Whether or not you have programmed before this module will give you the tools you need to perform research, complete assignments and most importantly will give you marketable skills that can help you get a job after your degree. If you're anything like the authors of this document it will also have the unintended consequence of giving you an **unhealthy coding addiction**, which will be impossible to kick.

Module Structure

This module is run in tandem with Physics In Practice (PIP) but don't be fooled, it is just as important as many of the full fat modules you'll do in your degree. As such, it is structured in much the same way as your other modules with teaching, assignments, and exercises within the lab book. The main difference is in **how** the material is taught; rather than traditional lectures you'll take part in lab sessions and work through this document chapter by chapter. These lab sessions, starting with a very short introductory talk, will give you direct contact with the AT and the lecturers to help you with any problems.

In addition to these sessions we will record and upload any relevant short lectures as videos to canvas to be digested at anytime. We also would like to draw your attention to the discussions tab on canvas, here you can post questions and one of us will get back to you as soon as we can or we can plan a full explanation for one of the workshops.

Each chapter in this document is typically a single lab session; this translates to a single chapter per week. The exceptions to this rule are weeks 4, 7, and 10 in which you will have an opportunity to work on assignments. The first (week 4) assignment is formative and will be peer marked. We will give you further details on this marking in the introduction sessions. The second two assignments will be tutor marked and contribute equally to the portfolio mark, worth 30% of the PIP module mark.

We will repeat this again and again: do not take the low weighting of this sub-module as a reflection of it's importance, `Python` is an essential skill. Failure to engage now **will** lead to troubles down the road when the weightings are far higher and contributory. We have seen it before and, sadly, will undoubtedly see it again where a student hits a **Python** related problem in a later module which was discussed during this module.

Lab book

You will find the entirety of this lab book uploaded to canvas (any updates to the content will be uploaded when necessary) in case you wish to move through the course faster than planned. However, not everyone enjoys coding as much as we do, **so do not worry if you find yourself struggling a little**, please make sure to ask the lecturers or AT for help if you need it. Programming can be daunting for the uninitiated and it can take time to start thinking like a programmer, but when you do your increased understanding of logic will help you throughout your whole degree/life.

Each chapter will begin with a list of learning outcomes for that section followed by the content containing some worked examples. At the end of each section, there will be some exercises for you to complete to check your understanding of the chapter (remember to ask the ATs for help if you're stuck, they will **likely** have made all the same mistakes).

Throughout this lab book you will find boxes like this. These contain information which is beyond the scope of this module but will be important should you decide to continue into computationally heavy research or a career using `Python`. They will be useful for future reference, or are just interesting, but don't be intimidated if you don't immediately understand the content contained within them.

Assignments

There will be three assignments, the first of which is unweighted and will be peer marked, and they will be marked on:

- The final answers you have.
- Coding style (including comments!!! More on this in Chap.4).
- How logical and efficient your approach is.

The assignments will be uploaded to Canvas and should be submitted by the deadlines in weeks 4, 7 and 10. These assignments, like all others, should be completed and submitted on your own, containing only your own work (a reminder about collusion and plagiarism is provided in section 5.2.1).

Cautionary Note

This module has been redesigned to use `Python-3`. This is the newest version of `Python` (replacing `Python-2`) and the version that's installed on the University system. You may come across instances of `Python-2` throughout your degree (and beyond) since it is still widely used but official support has now ended. There is little difference between the two versions on the surface, so don't worry about this too much. The main differences are the syntax of the `print` function and how the code behaves when dividing one integer by another.

The `print` function (as you may have guessed), prints a value to the screen. Should you be interested, the exact differences between the `print` function and integer division in each `Python` version are as follows:

Python-2

```
In [1]: print 'Hello World' # print statement, notice the lack of ()
        1 / 2 # integer division
        1.0 / 2.0 # float division
```

```
Out[1]: 'Hello World'
        0 # 0 since 1 is not divisible by 2
        0.5 # float division gives the expected result of a half
```

Python-3

```
In [2]: print('Hello World') # print function, notice the parentheses
        1 / 2 # float division is automatically assumed in\texttt{Python
    ↪ -3}
        1 // 2 # integer division now has a different operator
```

```
Out[2]: 'Hello World'
        0.5 # float division
        0 # integer division
```

Chapter 2
Some Useful Terminology

This is intended to be a quick point of reference for any terminology that appears in the script that may need clarification. Please feel free to suggest additions! This will evolve over time to contain anything and everything useful.

Data	This can be anything: a number, a set of numbers, a character, a set of characters, an image etc. Essentially any outside input you want to handle with `Python` could be called "data".
Data type	The variety of a particular data, kind of self explanatory but here for completeness.
Data structure	Any container for data. Think of a table or list in the real world, this is where you put *things*. In simple terms a "structure" in which you store "data".
Variable	These are labels that you store things in. Not to be confused with a data structure, a variable essentially labels a value, a data structure or an object and allows the computer to store and use what you have labelled. They are much like variables in maths, m representing a mass, F representing a force, c representing the fundamental speed of light. What the computer actually does is link the label (variable) to a memory address at which your value/object is stored.
Comment	A statement within a piece of code not executed by the computer but instead included to provide context to a human reader of the code.
Function	A reusable segment of code defined by the programmer which contains a set of operations, taking in values and returning the results after execution of the commands within the function.
Argument	A value that is passed into a function to be used inside the function. These can be anything your function needs to operate. Also often called a parameter but rarely in `Python`.

Module	Simply, a module is a file consisting of `Python` code which can be loaded in your `Python` programs to add functionality not native to `Python`.
PEP-8	This is a set of guidelines for writing good "Pythonic" code which makes it easier to read other peoples code.
IDE	(Integrated Development Environments) A piece of software that provides everything a user needs to write code; think Word but for code development.
Keyword	Keywords are the reserved words in `Python` used to perform specific operations. We cannot use a keyword as a variable name, function name or any other identifier as this would overwrite the python native behaviour of the word.

Table 2.1: Some useful terminology for the module.

Chapter 3
Jupyter: Quick reference

Jupyter Keyboard Shortcuts

This section is meant to act as a quick reference for the most useful shortcuts in `jupyter`, it'll become more useful when you've become more familiar with the software. It is worth trying to remember these shortcuts, as they can greatly improve the speed of workflow, and are just generally more convenient.

General Use:	
shift + enter	Run Current Cell
alt + enter	Run and insert new cell below
ctrl + S	Save Notebook
Command mode shortcuts:	
esc + B	Insert cell below
esc + A	Insert cell above
esc + D,D	Delete current cell
esc + ↑	Go to the cell above
esc + ↓	Go to the cell below
esc + C	Copy the current cell
esc + X	Cut the current cell
esc + V	Paste cell
Cell editing shortcuts:	
tab	Indent or code completion
ctrl + A	Select everything in a cell
ctrl + ?	Comment out the current line

Table 3.1: Some of the most useful `jupyter` keyboard shortcuts.

Chapter 4
The Basics

4.1 Learning Outcomes

By the end of today's session you will know how to:

1. Generate a simple flow chart, to understand their use in logic problems.
2. Run `Python` on the university computers using the `jupyter` interface.
3. Interrupt a (pseudo) infinite loop.
4. Use the `print` function.
5. Use `Python` as a simple calculator.
6. Use three different `Python` data types: integer, float, and complex.
7. Add comments to your `Python` codes.

4.2 Preamble

We'll start slowly to make sure everyone feels confident with the absolute basics. We'll start by showing you how to get started using `jupyter`, a web browser application that is one of several ways of coding in `Python`.

You will then work through some simple calculations and work with data types and variables. You are free to move on to future chapters if you complete this quickly or already have an understanding of `Python` programming.

Lets start by making a flow chart to describe the following:

Something that includes a series of steps that you do so often that it is second nature (don't be rude/personal/gross! Shouldn't need saying but past experiences...), e.g. making a cup of tea, having a shower, getting the bus to campus.....

These flowcharts should be done by hand initially, but if you are keen and want to learn a new skill, you could try making them using online flowchart software such as `lucidchart.com` or `gliffy.com` or `drawio.com`. Please note that not all of these packages work on

Edge/Internet Explorer (try Chrome instead). These flow charts do not have anything with Jupyter. The intention is to get you thinking about how programming works, not to get you to write a code.

Each year we are asked "Why are we doing flowcharts?". So here is the answer: To fully understand the operations of a program, flowcharts are indispensable tools. In simple terms, programs operate by stepping through a set of instructions/decisions (the code) and executing them one at a time (just as you step through a flowchart). Although the examples above are simple, programs can get very complicated (astronomical survey analysis, data reduction at CERN, cosmological simulations). You can have tens of thousands of lines of code with multiple paths for data, a structure too complicated to just keep in your head. Flowcharts give a schematic view of the different paths data can take through these programs, and the possible decisions based on that data. Flowcharts get you to think about **how a program functions** before you've even written it, and are **an important skill** you may need later in your career.

Simple example flow charts can be found in the appendix of one of this document's author's published papers (Roper et al. 2020), mentioned here just to drive home that they are useful!

4.3 Running `Python` on the University computers

4.3.1 Running `Anaconda` for the first time

The first time you use Anaconda on a University PC, you need to launch Anaconda via the Software Hub. To do this, type 'software hub' into the search box at the bottom left of the screen, and run the *Software Hub* app, see Figure 4.1:

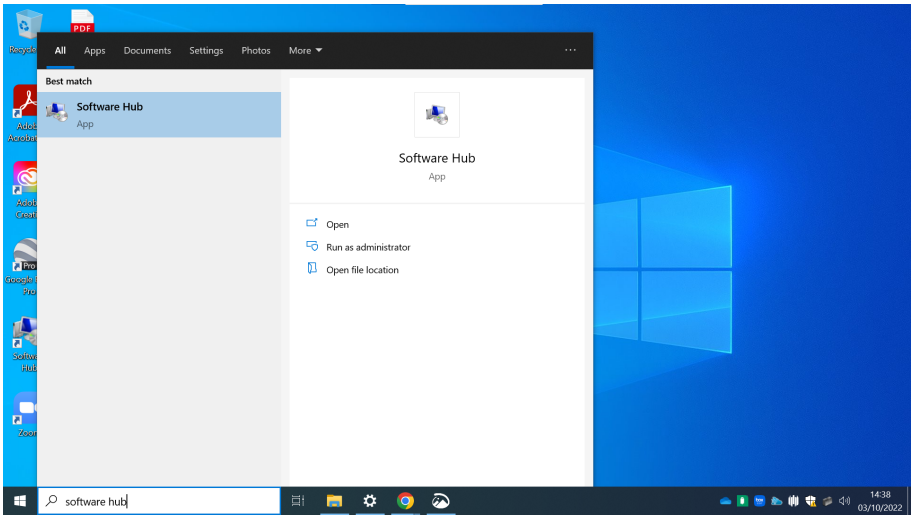


Figure 4.1: Running Software Hub (first use of Anaconda).

Type anaconda into the software hub search box and click Launch (Fig. 4.2).

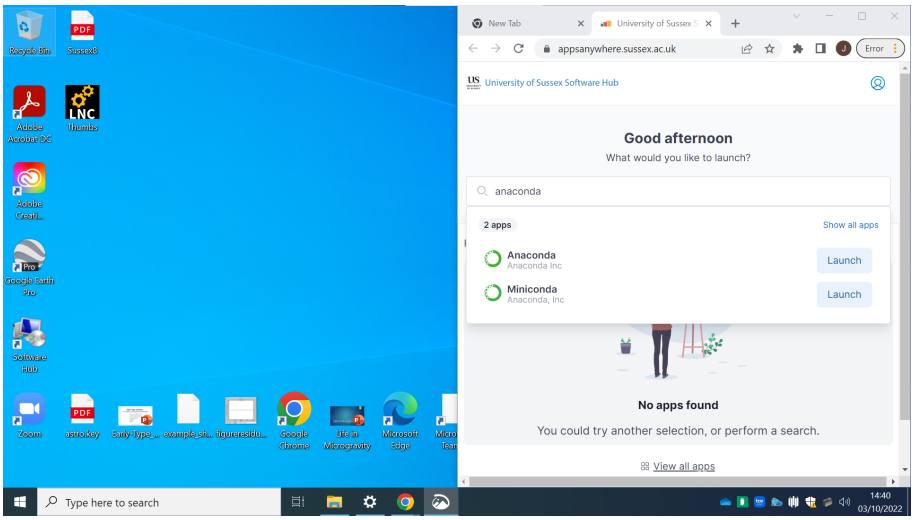


Figure 4.2: Launching Anaconda (first use).

4.3.2 Running **Anaconda** in subsequent sessions

On subsequent use, you should only need to select Anaconda from the Windows `Start` menu, see Fig. 4.3.

4.3.3 Running Jupyter Notebook

After a few moments, the Anaconda Navigator will appear. From it, launch Jupyter Notebook (Fig. 4.4).

Important: any Jupyter notebooks you create should be stored under the folder `OneDrive - University of Sussex`, otherwise they will only be saved on the particular PC that you are using. Open the `OneDrive - University of Sussex` folder and create a new sub-folder with a name such as `PIP_Python`. Open that sub-foldder and create a new Jupyter notebook using the `New` → `Python 3` pull-down menu, see Figures 4.5 and 4.6.

4.4 Running **Python** on your own computer

Throughout this term we will be using `jupyter notebooks`. `jupyter` is a web-browser application that allows you to create and share programs, which in our case will be written in `Python`. This section intends to help you install `Python` on you personal computer. Should you need to use `Python` on University computers rather than a personal computer the usual guide for this module can be found in Sec. 4.3. It is worth giving this a read anyway even if you are using a personal computer as it contains some important information.

The easiest way to setup everything you will need for the assessments and labs on your laptop or home PC, is by installing the Anaconda software package (`https://www.anaconda.com`). Anaconda is a 'distribution' of `Python` that has several extra features that make it easier to improve `Python`'s capabilities; one of the most important is its package manager.

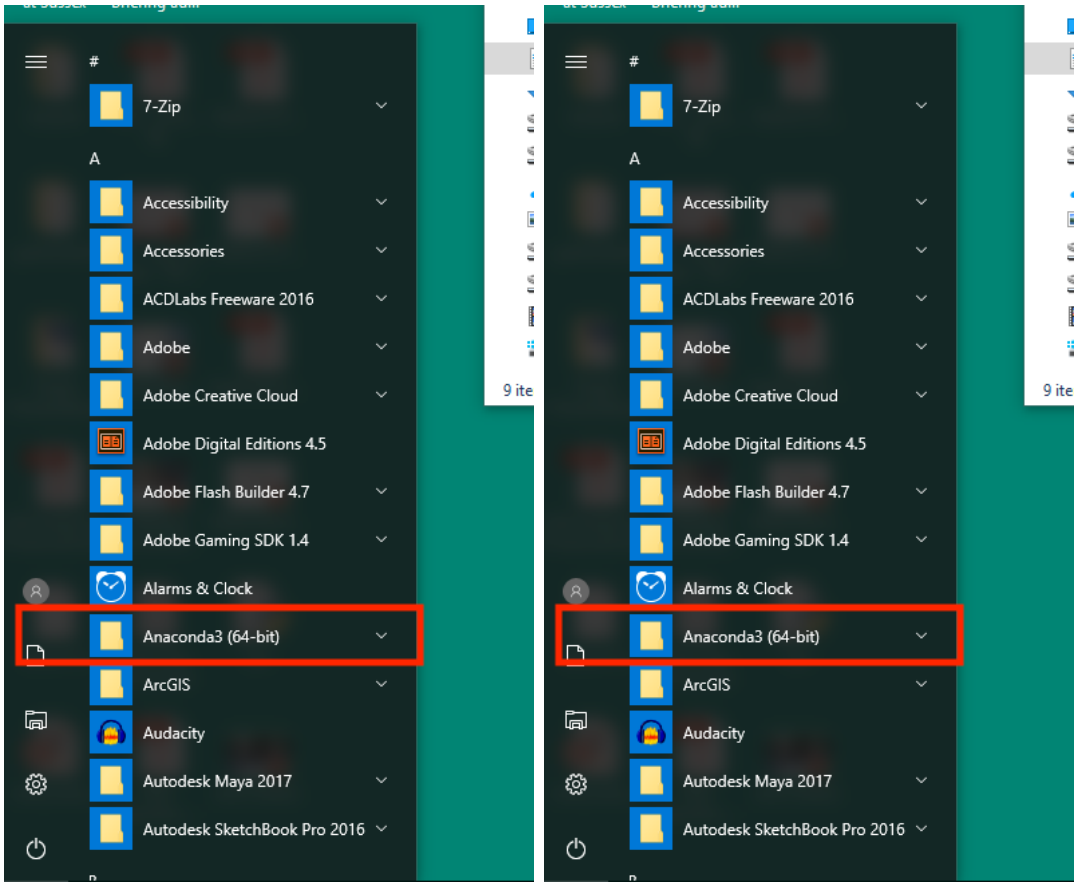


Figure 4.3: Running Anaconda, subsequent use. The program menu can be found in the bottom left corner

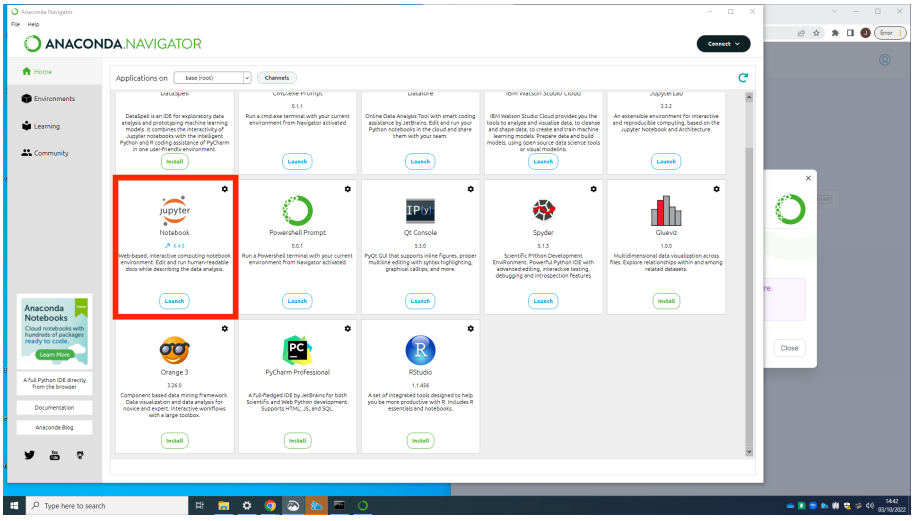


Figure 4.4: Launching Jupyter.

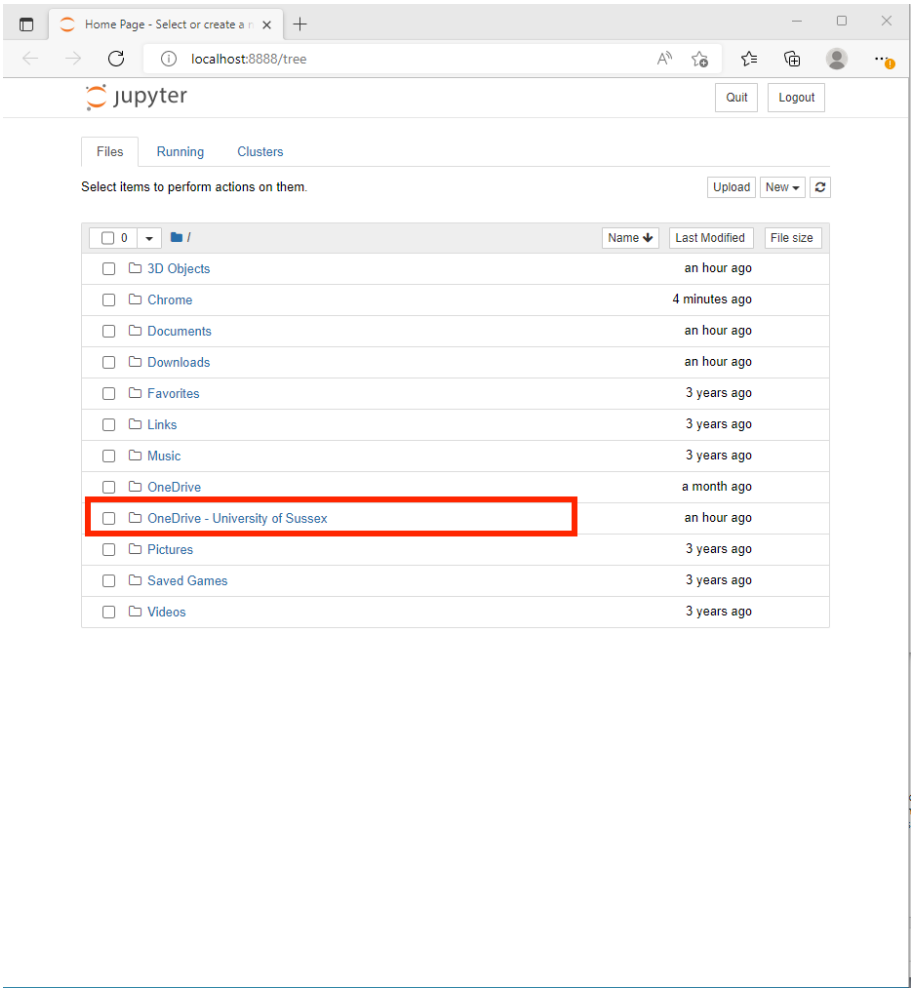


Figure 4.5: Be sure to save notebooks under `OneDrive - University of Sussex`.

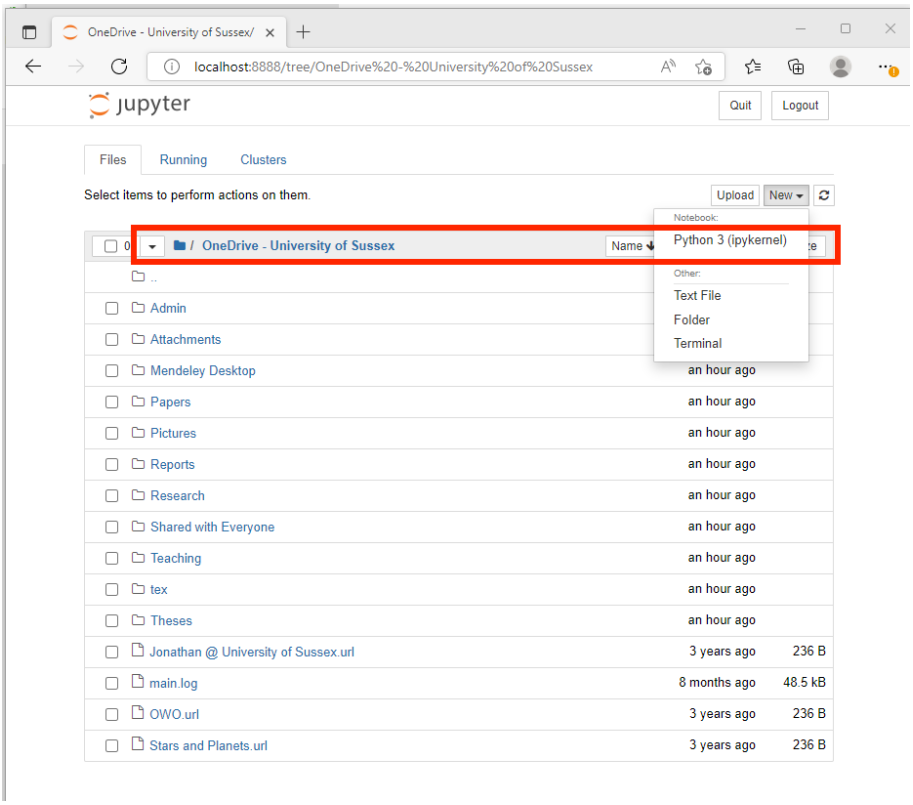


Figure 4.6: Launching Python.

When you reach chapter 7, you'll be introduced to **modules**, which are pieces of code that other people have written to expand `Python`'s capabilities. For instance, there are specialist modules for Astrophysics that make it easier for us to do common calculations and operations, such as loading in images from telescopes. `Python` has a built in way of installing new modules called `PIP`, but Anaconda adds a more sophisticated package manager which makes sure that all of the modules are the right versions and will work together (amongst other things).

Once you've opened the website, you should:

1. Click the download button, at the top right of the website.

2. Scroll down and select your operating system.

3. Choose the version you want, **for this course it should be `Python` 3.6 or greater**. Should you find yourself using the University system, the version installed there is `Python` 3.9.

4. I'd advise the graphical installer for the inexperienced. Windows users must find out what version of Windows is installed, 32 bit or 64 bit. Go to Control Panel - System and Security - System, and you'll find the answer in the system type field.

5. Now you can download the correct version of Anaconda and install it in the usual way! (Windows users will asked if they wish to add `Python` to their PATH, and if you want to be able to run `Python` from command line then you will need to).

6. When you've installed Anaconda, you'll be able to open Anaconda Navigator.

7. You'll be presented with several different ways of interacting with `Python`, if you select Jupyter Notebook (**not JupyterLab**), then you'll be presented with the interface you use in the lab sessions.

If you do your assessments on a personal machine **please ensure that you installed the correct version of `Python`**. If you fail to check this and your code doesn't work when we try to mark it then I'm afraid its all on you!

4.5 "Hello World"

Into your fresh new `Python-3 jupyter notebook`, type `print('hello world')` into the console and then press the "play" key. The result should look like Figure 4.7. Note that there is a keybord short cut for "play": ctrl + ↵ (`control+Enter`). For a reference table of useful keyboard shortcuts, look at page 1.

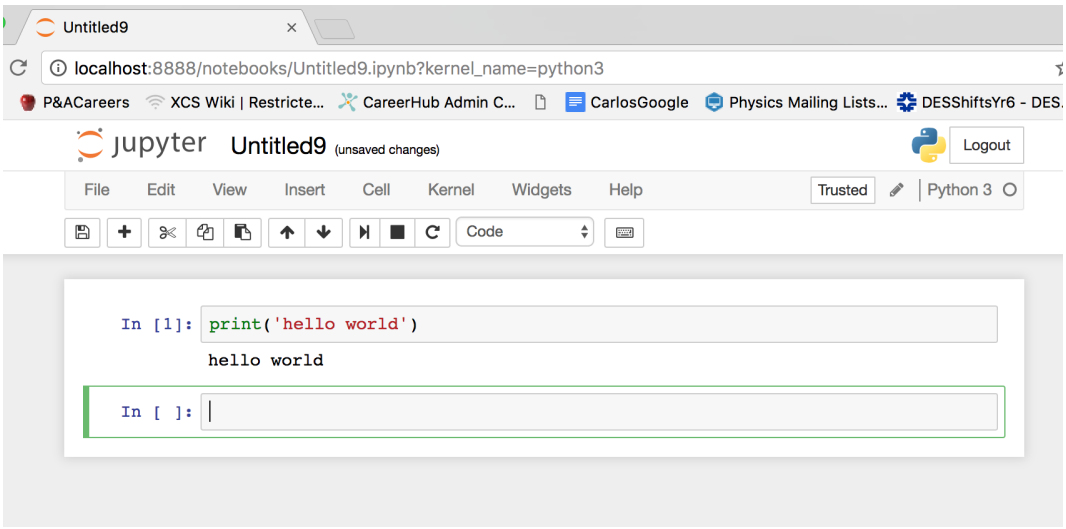


Figure 4.7: Saying hello to the world using `jupyter`. Those parentheses and quotation marks are essential: try using `print` without them.

4.5.1 Saving and logging your work

Make sure to save your work regularly, either by pressing the save button found directly beneath the "file" tab, or by pressing ctrl + s (`control+S`). You can then load that file later to start back where you left off. Over the course of the term you will build up a lot of these files, so give them sensible names (e.g. `LabSession1.ipynb` etc.). See Figure 4.8 below.

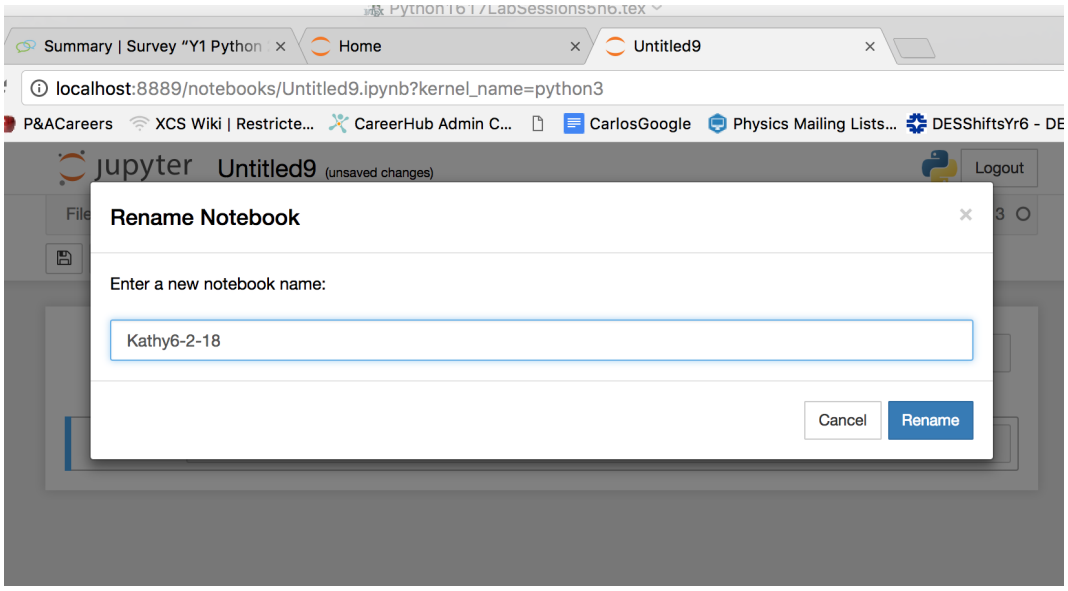


Figure 4.8: Your notebook will have a default, not very useful, name, such as "Untitled9". It is good practice to rename it to something that will help you remember what is in it. **Note:** Try to not use spaces in filenames as, although it works on Windows, Linux based systems will not like it - use an underscore instead (a rule to live by).

4.5.2 Infinite loops happen... and are very bad

If you suspect your code has gone into an infinite loop (a series of instructions that will continue forever), then you need to stop it immediately. Otherwise it may crash your computer (the first year this module was taught, the entire ITS network was disrupted by an infinite loop during a lab session). In `jupyter`, you go to the "kernel" tab and click "interrupt". Commit this to memory: infinite loops happen to everyone from time to time.

If you find your screen becomes slow or unresponsive during the infinite loop - you can also double tap "i" on the keyboard when a cell is not selected - this should also interrupt the kernel.

4.6 Data types in `Python`

In coding, variables are used to store information and give it a descriptive label (the variable name). `Python` variables can have many different data types, the most common of which are:

Integer - These are just integer numbers ($\mathbb{Z}$) such as 1, 53, 8651247 etc.

Floats - These represent the real numbers ($\mathbb{R}$) to a certain precision. Integers, fractions and irrational numbers can all be stored as floats. E.g, 2.0, 12.5, 6.3215314, π etc.

Complex - These are complex numbers ($\mathbb{C}$), in the form: a+bi, 3+5i, 7-4i etc. You should have used these in your maths course.
Note: In `Python` the complex number i is represented with j, following the engineering convention.

Strings - These are a sequence of characters. You set the beginning and end of the sequence by enclosing the characters in quotation marks, e.g. 'Hello', "F", 'this is also a string'.
Note: The quotation marks can be double or single. Choose a convention and stick with it, due to apostrophes it can be best to get in the habit of using "double quotation marks".

Boolean - These represent the two values of Boolean logic, True or False.

Lists - These contain multiple elements and can be different data types. They are also **mutable** meaning they can be edited after definition.

Tuples - Tuples are the same as lists in most respects. However, they are **immutable** meaning that once created they cannot be changed.

Dictionaries - This data type contains a set of key:value pairs, with the value being mapped to a *unique* key. Like lists, they are mutable. These are one of the most powerful tools in `Python` and will be covered in a later practical session.

During this first lab Session, you'll only need to work with integers, floats and strings. We'll work with the rest later in the term.

4.7 Simple calculations with `Python`

With integers, floats and complex numbers we can use `Python` to perform calculations. Here are some examples. Try them all in your `jupyter` notebook. In the following examples `In`'s and `Out`'s are included to emulate the `jupyter` output but please note your numbers accompanying them may differ on your screen. Normally we would need to use the print function to see the output (as in section 4.5), but as you are using `jupyter` and only doing one calculation per cell the result will appear at the bottom.

Adding two integers and complex numbers (add a new cell by clicking the + button, beneath the "File" tab):

```
In [1]: 14 + 3
```

```
Out[1]: 17
```

```
In [2]: 5 + 2j + 4 - 9j
```

```
Out[2]: (9 - 7j)
```

Subtracting two integers.

```
In [3]: 32 - 8
```

```
Out[3]: 24
```

Multiplying two integers.

```
In [4]: 7 * 9
```

```
Out[4]: 63
```

Computing exponents using **.

```
In [5]: 2**3
```

```
Out[5]: 8
```

```
In [6]: 25**0.5
```

```
Out[6]: 5
```

You can also combine any of these operations. The rules regarding the order of execution of mathematical operators are the same as you're used to; multiplication and division happen before addition and subtraction etc.

```
In [7]: (4 * (723 + 67)) - 634
```

```
Out[7]: 2526
```

All of these calculations also work with floats (decimal numbers). `Python`-3 uses float division (normal division) by default, therefore when trying out the following calculation we expect the output to be 4.5.

```
In [8]: 27 / 6
```

```
Out[8]: 4.5
```

The alternative is integer division, where the result of the calculation is rounded down to an integer. This was the default in `Python`-2, and caused a lot of confusion for new programmers. Even though this no longer applies, it is always best to use floats explicitly, so that you get into good habits.

```
In [9]: 27.0 / 6.0
```

```
Out[9]: 4.5
```

```
In [10]: 27.0 / 6
```

```
Out[10]: 4.5
```

```
In [11]: 27 / 6.0
```

```
Out[11]: 4.5
```

Python-3 can still perform integer division with the following command

```
In [12]: 27.0 // 6.0
```

```
Out[12]: 4.0
```

To perform modulus division (i.e find the remainder) we use the following command.

```
In [13]: 42 % 5
```

```
Out[13]: 2
```

Here is a table containing the maths commands that come as standard with `Python`.

Table 4.1: Operators calculations

+	Addition
-	Subtraction
*	Multiplication
**	Exponent
/	Division (True)
//	Division (Integer)
%	Modulus

Now work out the following examples using `Python`.

1. 2 + 5
2. 4 - 5
3. 7×8
4. $(4 + 7j)$ - $(12 - 5j)$
5. $20 \div 8$ (integer division)
6. $20 \div 8$ (float division)
7. $56 \div 9$ (integer division)
8. 6^{22}
9. $(32 \times 7)^2$
10. $(3 + 8j) + (1 - 9j)$

11. $\sqrt{2}$
12. 0^0 (this should be troubling!)

> In fact, with regards to 0^0, you might not want this behaviour in your code. In that case, like many other odd behaviours in programming, you are trapped with how the programming language (mis-)behaves. You need to stay aware and code accordingly, and in extreme circumstances use error handling (discussed later).

4.8 Using variables for simple maths operations

We can also assign values to variables, and continue to manipulate them. If you feel a bit hazy about what we mean by a variable, its the x and y in an equation like $y = f(x)$, but in coding it could just as well be $b = f(a)$ or $banana = f(apple)$. Always try and give your variables useful (and slightly amusing) names, so you'll remember what they're for if you come back to a program after a break.

Here we assign 5 to a, 12 to b, -3 to c, and 10.4 to d, and then perform some calculations.

```
In [1]: a = 5
        b = 12
        c = -3
        d = 10.4
        a
```

```
Out[1]: 5
```

```
In [2]: b
```

```
Out[2]: 12
```

```
In [3]: c
```

```
Out[3]: -3
```

```
In [4]: a + b + c
```

```
Out[4]: 14
```

```
In [5]: a - c
```

```
Out[5]: 8
```

```
In [6]: b + 7
```

```
Out[6]: 19
```

We can now perform float and integer division discarding the remainder.

```
In [7]: print(a / 2)
        a // 2
```

```
        2.5
Out[7]: 2
```

We can also assign the result to a new variable, and use that for further calculations.

```
In [8]: r = a / 2
        r - a // 2
```

```
Out[8]: 0.5
```

This style of output is specific to this cell based interface with `Python` (found in `jupyter` and another software package called `iPython`). These will print the contents of **the last** variable or computation of a cell without the need for a `print` statement. If you find yourself running a `Python` script from a .py file or using another Integrated Development Environment (IDE, some are described in section 4.13) then this will no longer print an output. Other IDEs and `Python` scripts will also be run as a whole, and not individually cell-by-cell as in `jupyter`.

4.8.1 Test your understanding

Now carry out this task: Define the three variables `day_me`, `month_me`, `year_me` according to your birth date. Then define the three variables `day_now`, `month_now`, `year_now` according to today's date and then figure out how many days you have been alive. This doesn't need to be exact; aim to get within ±200 days. You can check your answer using one of the many online tools available (use Google to find them).

By now you should have seen that it is more efficient to write several lines of code into a single `jupyter` cell, and you can use the ⏎ (`Enter`) key to move to a new line. If you need to create a new cell you can either press the + button under the "File" tab or use the keyboard shortcut ⌥+⏎ (`Alt-Enter`), which will run your current cell before making a new one.

4.9 Changing A Variable's Data Type

You may find yourself in a situation where you need to change the data type of a variable. If we need to convert `a` to a float we can simply use the `float` function.

```
In [1]: print(a)
        float(a)
```

```
        5
Out[1]: 5.0
```

Conversely, if we need to convert to an integer we can use the `int` function.

```
In [2]: int(d)
```

```
Out[2]: 10
```

The value of a variable is not changed by converting it to another data type:

```
In [3]: print(float(a))
        a
```

```
Out[3]: 5.0
        5
```

If we wish to convert `a` permanently into a float we need to assign it to a new variable, this could be called anything but naming it `a` overwrites the old value stored in `a`. We can then proceed with the new variable `a` containing a float.

```
In [4]: a = float(a)
        print(a / 2)
        a
```

```
        2.5
Out[4]: 5.0
```

It's important to understand that the `int` function **does not** round a float up or down, it just truncates it. Lets demonstrate this by defining `d` so that it would round up to 11.

```
In [5]: d = 10.55
        d
```

```
Out[5]: 10.55
```

```
In [6]: print(int(d))
        d
```

```
        10
Out[6]: 10.55
```

If we wish to actually round a number then we can use the `round` function.

```
In [7]: e = round(d, 1)
        e
```

```
Out[7]: 10.6
```

The second argument is the number of digits to round to (arguments are values that you pass into a function, here the second argument is 1 and the first is the variable to be rounded). If you didn't give `round` a second argument, it would round to the nearest whole number and give you the result as an integer.

```
In [8]: round(d)
```

```
Out[8]: 11
```

4.10 Commenting

Comments are added to code to explain the function of lines in a way that a human can understand. They are essential for any project that's more than a few lines long, especially if you're working with other people in a professional setting or you expect to use this code again in the future. Other people may need explanations of code that you think is self-explanatory, and you may not remember how a project works when you come back to it in 6 months.

When writing code just ask yourself this question: Is it immediately clear what a line of code does? (e.g. `x + 2`, `print('Comments are great!!!')`, etc.) If the answer is no, or you're not sure, then put a comment!

We will be repeatedly telling you to add comments, both during the workshops and the assessments. Comment quality will actually factor into your assessment marks, so get into good habits now. In the last cell of the previous section the different methods for getting an integer were marked with **comments**. In `Python`, comments start with a hash (#) and are ignored during the execution of the program. Comments can appear anywhere in a line and anything that follows the # is 'commented out', and won't be run as code by `Python`.

There are two types, inline comments and block comments. Comments should take the general form:

```
In [1]: # This is a block comment describing the following block of code
        # I am going to do something because...
        the code that does the something you're doing...
        something particular  # an inline comment about this particular line
```

From this point on the examples will have useful comments, though feel free to call us out on any that you think we could do better.

4.11 Testing the type of a variable

Finally, we don't just assign integers and floats to variables, we can also assign strings and any other `Python` datatype. Also, variable names don't just have to be letters, they can be words too (as long as the word does not match any of `Python` built in functions like `print`, or `int`).

The fact that variables can be multiple different data types means you may need to find the type of a variable. You can test a variables type very simply using the `type` function, with this function the output returns of `str` and `int` stand for string and integer respectively.

```
In [1]: # Define variables to demonstrate type testing
        f = 45  # integer
        g = 5.2341  # float
        h = 3 + 5j  # complex
```

```
        type(f)  # test variable f
```

```
Out[1]: int
```

When combined with a print statement the `Python` object itself is printed. This object is an instance of the float class. If this is jibberish don't worry, it is mostly unimportant for the entirety of this module but plenty of resources exist online for learning about `Python` classes.

```
In [2]: print(type(g))  # test variable g and print the result
```

```
        <class 'float'>
```

Note that with a print statement `jupyter` omits the Out[...], mostly unimportant but don't be confused by it's absence.

```
In [3]: type(h)  # test variable h
```

```
Out[3]: complex
```

4.12 A note on the horrors of the insert key

Every year the same issue comes up, it is very easy to inadvertently press the insert key (insert). At first it will appear that nothing has happened, but if you try to type anything within text you've already written, you will start to overwrite the existing characters.

Say you are typing out a print statement,

```
In [1]: print('I want to prit stuff')
```

but here you have mistyped and accidentally forgotten the 'n' in print! So you go back to edit, if the insert key is on you would get

```
In [1]: print('I want to prin stuff')
```

as the 'n' has overwritten the 't'. If you encounter this behaviour then insert is the cause, simply press insert and problem solved.

4.13 Integrated Development Environments

A loose definition of an Integrated Development Environment (IDE) might be 'a piece of software that provides everything a user needs to write code'. IDEs exist for every programming language, and normally include (at the very least):

1. A source code editor - Lets the user write and edit their code.
2. A debugger - Tools to check for issues with the code.
3. A compiler - This assembles and runs the code. Different types of programming language approach this differently; languages like C++ have to be compiled (like assembling a machine before you use it) into an 'executable' before they're run. `Python`, on the other hand, is an interpreted language, and is executed line-by-line (like reading a recipe and doing as instructed on each line).

There are a huge number of IDEs to choose from, especially for such a popular language as `Python`, and ultimately you just have to try a few until you find your favourite. We will summarise a few of the most popular options here, and if you're interested in the subject you can do your own research. Many IDEs are free, but those that do charge often have a student discount option, so look out for that.

- Text Editors - Although not technically IDEs, text editors like Notepad and Gedit get an honourable mention as you can write code in them and then run it from the terminal. For instance, you could write `print("hello world")` in a file called test.py, save it, and then run it by typing `python test.py` in terminal.
- Jupyter Notebook - The interface you're familiar with, a relatively lightweight IDE without some of the more complex (and professional level) features you may find in other pieces of software. Its characterised by the cell based approach to coding you will have seen.
- Spyder - A more traditional `Python` IDE than Jupyter, with some more features. If you install Anaconda on your own system, it is included in Anaconda Navigator.
- JupyterLab - This is a less lightweight version of Jupyter Notebook, with the same cell based system but more powerful features. Consider it a cross between Jupyter Notebook and Spyder, it is also included in Anaconda Navigator.
- PyCharm - This is my personal favourite IDE, and has basically every feature you can ever imagine needing. There is a free version and a paid for professional version (though students can get both for free). It requires a relatively good computer to run it, otherwise it can be a little laggy.
- Atom - An infinitely customisable bit of software, whose capabilities can be expanded and adapted by using plug-ins written by other users.
- VS Code is also recommended.

That's All Folks!

You've completed the first week. See you next time!

Chapter 5
Strings, Booleans, Tuples and Lists

5.1 Learning Outcomes

By the end of week 2, you should be able to:

1. Still do everything described in the script for lab session 1.
2. Manipulate strings.
3. Use Boolean variables.
4. Use tuples and lists.
5. Use the `input` command.

5.2 Preamble

Today you'll work through several sections that introduce various features of `Python`. Typically there will be one or more exercises at the end of each section; make sure you work on these before moving on. Ask for help if you're unsure or get stuck, its what we're here for!

5.2.1 A reminder about collusion and plagiarism

Don't do it. By all means work together on the exercises during the lab sessions, and by all means *talk* to each other about ways to approach the assessments. However, if find yourself de-bugging a friend's code for an assessment, you've gone too far - you've got to let them figure this stuff out themselves. If you find yourself cutting and pasting from a friend's code, then that's even more serious. If incidences of plagiarism and collusion do come to light, those involved will be referred to the appropriate disciplinary committee.

5.2.2 Sourcing Help

When coding, in this module or in the future, you may run into inexplicable errors or wonder what the best approach to a problem is. By all means ask the ATs for help but you should also be aware of `www.stackoverflow.com`, a community driven forum where people post their

problems and users attempt to provide solutions. If you run into a problem it's almost certain that somebody else has already had it, and asked about it somewhere on the internet, so it's worth having a look. As an aside, if you get a large, confusing error message in a Jupyter notebook, you can look for a -> symbol near the top, and that will show you the line of code that failed to execute.

5.3 String manipulations

Now on with today's session. In the previous chapter you were introduced to strings, a data type containing characters. Strings can be manipulated very easily with inbuilt operators (literally a symbol that performs an operation on a variable) such as `'+'`, which can be used to add strings together (concatenate them). Strings can be 'indexed' to extract specific characters or sliced with the `':'` operator to extract a set of multiple characters. **Note that `Python` starts counting elements from zero**, which means that the first character in a string is character zero. Here are some examples, try then all in your `jupyter` notebook.

We can add strings to concatenate them.

```
In [1]: # Define strings for manipulation
        s1 = "This is a"
        s2 = "string"

        print(s1 + ' ' + s2)
```

```
        This is a string
```

We can index a specific element by indexing with square brackets (note: an element of a string includes characters and **also** whitespaces). This is also a good time to point out that the `print` can print as many variables as you like, just seperate them with a comma:

```
In [2]: # Print the first and last element of s1 to demonstrate indexing
        print(s1[0], s1[-1])
```

```
        T a
```

Notice we can extract the final element by indexing with -1. In fact we can count back from the end of a string with n characters by using a negative index (i.e. -1 extracts the n^{th} character **a**, -3 extracts the n-2^{th} character **s**, -4 extracts the n-3^{th} character **i**, etc.).

We can also slice with indices by using the `:` operator, [start index:end index]. Bear in mind that `Python` indexing is inclusive of the start index and exclusive of the end index, so [0:4] would only retrieve characters 0, 1, 2, and 3.

```
In [3]: # Slice the string to get the first 6 elements
        print(s1[0:7])
```

```
        This is
```

```
In [4]: # Get the second to the 4th elements
        print(s1[1:4])
```

```
        his
```

Additionally strings can be multiplied by integers to make them repeat.

```
In [5]: # Print 'This' three times separated by spaces
        print(s1[:5] * 3)
```

```
        This This This
```

Final thing to note with slicing is that the 0 (first index) can be omitted as in the previous example. `Python` sees this as 'from the beginning', the same is true for the end if the second index is omitted (`[2:]`, i.e. the 3rd element to the end).

5.3.1 Exercises

Exercises (with solutions at the end of the chapter)

Try the following (for answers see Section 5.9)

1. (a) Assign the strings '`hello`' and '`world`' to two separate variables.
   (b) Using the two strings print out '`hello world`'
   (c) Print the word hello 20 times with no spaces using the assigned string.
2. (a) Declare a variable and assign the following string to it '`Do not push the red button`'
   (b) Slice the string to form '`Do not push`'
   (c) Slice the string to form '`push the red button`'

Exercises (other)

1. W2Basic1 - Create a string variable `myname` that is your full name - first, middle (if you have them) and last (family name).
   (a) Slice the string so that it prints your last name only.
   (b) Slice the string such that it prints your first name, middle initial and then your last name.
   (c) Print 'I am' and your name.

5.4 Boolean Variables

Boolean expressions represent the two values of Boolean logic, `True` or `False`. They can be used to compare variables, if the variables are the same then the result will be `True`, and if they are not then the result will be `False`.

Note: If the variables are of a different type (float and integer), but are the same number, the result will be `True`, for instance `1 == 1.0` will return `True`. This is not true if one variable contains a string and another a float or integer, `'1' == 1` will return `False`.

```
In [1]: # Define variables to demonstrate Boolean operations
        a = 12
        b = 5
        c = 7.8

        # Test if a is equal to b
        a  ==  b
```

```
Out[1]: False
```

```
In [2]: # Test if a is greater than or equal to c
        a  >=  c
```

```
Out[2]: True
```

Multiple boolean expressions can be tested at once using the keyword `'and'`.

```
In [3]: # Test if b is greater and c and b is greater than a
        print(b > c and b > a)
```

```
        False
```

Multiple boolean expressions can also be combined without `'and'`

```
In [4]: # Test whether a is greater than c and c is greater than b,
        # without using and
        print( b < c < a )
```

```
        True
```

Table 5.1 summarises the Boolean operators . [Note that you can also use `in` but only with tuples and lists, see Table 5.2.]

==	Equal to
!=	Not Equal to
>=	More than or equal to
<=	Less than or equal to
>	More than
<	Less than

Table 5.1: Boolean operators available in `Python`. Note that you can also use `in` to check if something is in a list or tuple, it'll return a Boolean.

5.4.1 Exercises

1. W2Basic2 - Assign 13 to a variable q, 2 to a variable w, and 6.5 to a variable e.
   (a) Check that q is less than w.
   (b) Check that $q \div 2$ is equal to e.

Note: for these exercises, make sure a `True` or `False` is displayed on screen as needed.

5.5 Tuples & Lists

The following data types are commonly stored in lists or tuples:

- Integers
- Floats
- Strings
- Boolean

In fact any `Python` object can be stored in a list or tuple. That includes other lists and tuples, so it is possible (and sometimes useful) to have a list of lists. As mentioned in section 4.6, lists and tuples are essentially the same; the key difference is that lists are mutable (editable) and tuples are not. Figure 5.1 shows the general structure of these objects.

Anatomy of Lists and Tuples

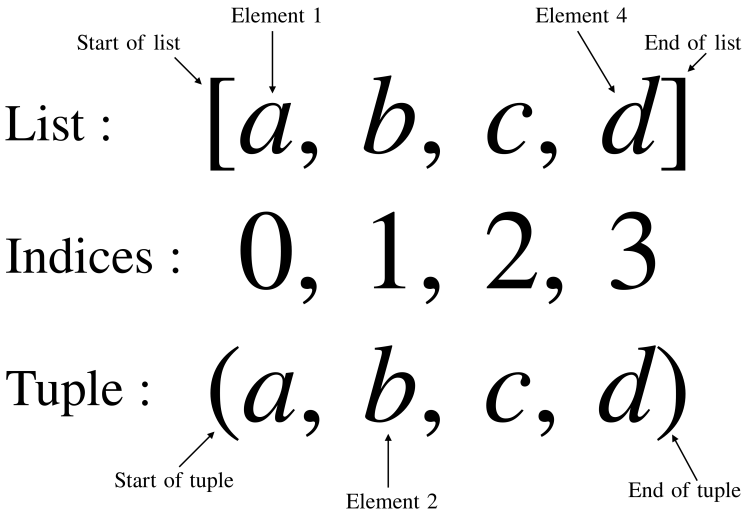


Figure 5.1: A diagram showing the anatomy of `python` tuples and lists. Lists, tuples and strings all have the same index conventions where the first element is the zeroth element. The way a list is differentiated from a tuple is the symbol used to denote the beginning and end of each; for a list `[]` and for a tuple `()`.

5.5.1 Tuples

Once a tuple has been declared, it cannot be changed or edited in any way. A tuple is declared using parentheses '`( )`' to surround the elements (the contents of the list), which must be separated by commas. You can also create a tuple containing only one element, by putting a comma after the first entry in the tuple (i.e. `(element1,)`).

```
In [1]: # Define a simple 1 element tuple
        simpletuple = (5, )

        print(simpletuple)
```

```
        (5, )
```

```
In [2]: # Redefine the simple tuple contain multiple elements
        simpletuple = (5, 4, 3, 2, 1)

        print(simpletuple)
```

```
        (5, 4, 3, 2, 1)
```

We can index a tuple just as we have indexed strings to extract individual elements.

```
In [3]: # Extract and print the third element in the tuple
        print(simpletuple[2])
```

```
3
```

Rather than indexing every time we want a specific element from a tuple we can also extract each element into its own variable, effectively naming that element of the tuple. Notice here the LHS has the right number of variables to extract each element into its own individual variable; if this is not the case an error well be raised telling you there were "Too many values to unpack".

What happens if you leave out the quotation marks around *Bruce*, and do you remember why?

```
In [4]: # Define a new tuple of strings
        name = ('Bruce', 'Batman', 'Wayne')
        print(name)

        # 'Name' (assign to a variable) each element of the tuple
        firstname, superHeroName, lastname = name
        print(firstname)
```

```
        ('Bruce', 'Batman', 'Wayne')
        Bruce
```

We can then use the variables containing individual elements.

```
In [5]: print("*Gruff Voice* I am " + superHeroName)
```

```
        *Gruff Voice* I am Batman
```

As with a string, we can slice the tuple to extract a range of elements, here the first and second. Lists can be sliced in exactly the same way.

```
In [6]: # Extract the first and second element of the tuple and print
        print(name[0:2])
```

```
        ('Bruce', 'Batman')
```

We can also combine all of these together along with some string slicing (remember what adding strings together is called and put it in a comment) .

```
In [7]: print("*Gruff Voice* I am " + name[0] + " " + lastname[0:2]
              + "... I mean " + superHeroName)
```

```
        *Gruff Voice* I am Bruce Wa... I mean Batman
```

5.5.2 Lists

Lists, unlike tuples, are mutable and can be edited; this feature means they tend to be used a lot more than tuples. Lists are defined by using square brackets '`[ ]`' with elements separated by commas (just as in a tuple).

Useful list methods (functions used by putting a dot after a variable) include:

- `append` - adds an element to the end of a list.
- `extend` - add a list of elements to the end of a list.
- `insert` - which allows you to add an element at a given position.
- `pop` - this removes and returns an element at a given position.

Another very useful function is `len`, which will return the number of elements in a list (this also works with strings, so `len("python")` would return **6**. Just like strings and tuples, lists can be indexed and sliced to extract certain elements.

```
In [1]: # Define a list
        alist = [2, 4, 6]

        # Append 10 to the end of the list
        # (add an element to the end)
        alist.append(10)

        print(alist)
```

```
        [2, 4, 6, 10]
```

```
In [2]: len(alist)  # Get the length of the list
```

```
Out[2]: 4
```

Here is an example of the use of the insert method compared to simply overwriting elements in the list.

```
In [3]: # Insert 8 into the 4th position (index 3) within the list
        alist.insert(3, 8)

        print(alist)
```

```
        [2, 4, 6, 8, 10]
```

```
In [4]: # Overwrite elements 3 and 4 (index 2 and 3) with strings
        alist[2:4] = ['Something', 'to']

        print(alist)
```

```
        [2, 4, 'Something', 'to', 10]
```

For use with strings, tuples, and lists:	
`a + b`	Joins items together
`a[i:j]`	Outputs elements from i to j-1 of a
`a[i]`	Outputs i^{th} element of a
`x * a`	Produces x copies of a
`len(a)`	Outputs length of a
For use with a tuple or a list only:	
`min(a)`	Outputs minimum value in a
`max(a)`	Outputs maximum value in a
`n in a`	Outputs true if element n is in a
For use with lists only:	
`a.append(x)`	Adds element x to list a at the end
`a.insert(n,x)`	Adds element x to list a in position n
`a.extend(x_list)`	Adds the elements in `x_list` to list a at the end
`a.pop(x_list)`	Removes and returns an element at a given position

Table 5.2: Operators for strings, tuples and lists

5.5.3 The `is` Keyword

As well as checking if variables are equal you can check if two variables are literally the same object with the `is` keyword. This can be useful when programs get complex and copies of lists get involved (more on this later).
With variables defined as integers, floats, or strings `is` behaves the same as ==, which was introduced in section 5.4 (also where the `a` and `c` variables were defined).

```
In [1]: # Define a new variable d which IS a
        d = a
        e = 12

        # Test if c is d
        print(c is d)

        # Test if a is d
        print(a is d)

        # Test if d is e and a is e
        print(d is e and a is e)
```

```
        False
        True
        True
```

Here it is obvious that `d` is `a` and `c` is not `d`. The important thing to note is that when using lists (or any `Python` variable more complex than a integer, float, or string) `is` will return `False` even when two variables contain the same elements. This is because the two variables are different instances of the `Python` object and are located at different memory addresses.

```
In [2]: # Define new variables containing the same list
        list1 = [1, 2, 3]
        list2 = [1, 2, 3]
        list3 = list1  # Making an alias!

        # Test if list1 equals list2 and if it equals list3
        print(list1 == list2, list1 == list3)

        # Test if list1 is list2 and if it is list3
        print(list1 is list2, list1 is list3)
```

```
        True True
        False True
```

If you **assign a list to a new variable** (this is called aliasing - like the example above) and edit the new variable you will be **editing the original**! To avoid this you can test using `'is'`. The same rules also stand for tuples, though remember they are immutable and can't be edited.

Essentially the linked copy is just another name (an alias) for the original variable. To make an independent copy you can use the slice operator (but omit the start and end indices to take the whole list) or you can use the `copy` method.

```
In [3]: # Define a list
        lst = ['Aliasing', 'causes', 'me', 'so', 'many', 'problems']

        # Create an alias for the list
        alias_lst = lst

        # Create a copy with each method
        lst_copy1 = lst[:]
        lst_copy2 = lst.copy()

        # Test if the list is equal to the alias and copies
        print(lst == alias_lst, lst == lst_copy1, lst == lst_copy2)

        # Test if the list is the alias and copies
        print(lst is alias_lst, lst is lst_copy1, lst is lst_copy2)
```

```
        True True True
        True False False
```

5.5.4 Exercises

1. W2Basic3 - Create a tuple variable 'top5' with 5 elements containing your favourite animals.
   (a) Check if 'dog' is in your tuple, such that it displays `true` or `false` (see Tab. 5.2).
   (b) Print the 3rd item in your tuple.

2. W2Basic4 - Create a list called 'mylist' containing 10 numbers.
   (a) Print the length of the list.
   (b) Print the minimum value in the list.
   (c) Print the maximum value in the list.
   (d) Add the number 14 to your list of numbers and print the new length.
   (e) Insert (using the insert function) the number 8 to the 9^{th} place in the list.
   (f) Print the elements 5 through 9 (i.e. 5 numbers) in your list. Note the last element printed should be 8.

5.6 Input function

The `input` function is used to get a response from the user of a program, this will be your first interactive use of `Python`! An example can be seen below:

```
In [1]: # Get the users name and assign it to a variable
        name = input('Name: ')

        print('Hello ' + name)
```

```
        Name: |...
```

The input function will print the provided string and then output will hang waiting for user input. If the user then writes Batman for instance, the rest of the code following the input statement will be executed having assigned the Batman to the `name` variable.

```
        Name: Batman
        Hello Batman
```

The `input` function will always return a string, so don't let this catch you out. If you need an integer input from the user you have to convert the string returned by `input` to an integer, like so:

```
In [2]: # Get a number from the user, convert to an integer
        print('Choose a number')
        num = int(input('Number: '))

        # Multiply this number by 5 and print the result
        print('5 times your number is:')
        print(5 * num)
```

which for the input of 5 results in

```
        Choose a number
        Number: 5
        5 times your number is:
        25
```

This takes your number and multiplies it by 5.

5.6.1 Issues with `jupyter`

Sometimes with the `input` function (though occasionally with others), you might see a `jupyter` cell that looks like this:

```
    In [*]:
```

and you are unable to get any outputs from any of the cells in your notebook. If this happens, essentially `jupyter` has frozen. Simply click on the "Kernel" dropdown menu and restart the notebook, and bear in mind that restart and clear all will remove all of your `Out` cells. Any form of restart will remove all previously defined variables from memory, meaning you'll have to run all your code again.

5.6.2 Exercises

1. W2Basic5 - Calculate a tip for a meal
   (a) Create a short set of commands (in one `jupyter` cell), that allows the user to enter the price of their meal in a restaurant.
   (b) Then print out a message displaying a price including a 15% and 20% tip.
2. W2Basic6 - Calculating your age in seconds
   (a) Create a short set of commands (in one `jupyter` cell), that allows the user to enter their age at their last birthday. Assign this value to a variable `age`.
   (b) Use the `age` variable to calculate their approximate age in seconds (ignore leap years).
   (c) Print 'You are over N seconds old', where N is their age in seconds..

5.7 Signpost

You should have at least reached this point by the end of the lab session in week 2. If you are here with plenty of time to spare, then you have several options:

1. Leave now (though get an AT to check that you've done enough first).
2. Start on the first assessment/competency test.
3. Work through some of the advanced problems in section 5.8 (**this is the preferred option**).

5.8 Advanced Exercises - Some independent learning may be required

1. Print out hello world so it looks like the this using only one print command:

```
Out [1]: Hello
         World!
```

   Hint: Think invisible characters...

2. Given the diameter of the sun is 1,391,000 km, print out the time in years (as an integer) it would take to drive round the sun's circumference at 100 kilometres per hour. (Take π as 3.141)

3. Create a string variable called `animal` and assign an animal to it. Using one print command print `animal` on five different lines.

4. Create a list called colours, containing two elements 'red' and 'green'.

   (a) Add 'blue' and 'yellow' to the list, put it in alphabetical order, and print it.

   (b) Remove 'yellow' from the list.

   (c) Print the list in reverse.

   (d) Make an independent copy of 'colours' called 'RGB'

   (e) Add 'yellow' back to colours.

   (f) Print '`colours=`' and '`RGB=`', note: `colours` should contain one more entry than `RGB`.

   (g) Show (using a Boolean) that 'colours' and 'RGB' are not equal in length.

5. Create a tuple called 'Words' containing 5 words of your choice and print the item.

   (a) Using the slice operator, slice words such that the second and third words are printed.

   (b) Find an alternate slicing method to produce the same result as above.

5.9 Worked Solutions for Section 5.3.1

```
In [1]: # Worked solution for Week 1, Exercise 1a
        # Define two string variables
        first_word = "Hello"
        second_word = "World"

        # Concatenating, with a space, and assigning to new variable
        full_sentence = first_word + " " + second_word

        # Printing the finished phrase
        print(full_sentence)
```

```
        Hello World
```

```
In [2]: # Worked solution for Week 1, Exercise 1b
        # Printing out the string 20 times
        print(first_word*20)
```

```
        hellohellohellohellohellohellohellohellohellohellohellohellohellohello
        hellohellohellohellohellohello
```

```
In [3]: # Worked solution for Week 1, Exercise 2a
        # Define the string variable
        warning_phrase = "Do not push the red button"

        # Printing the warning phrase, because I can
        print(warning_phrase)
```

```
        Do not push the red button
```

```
In [4]: # Worked solution for Week 1, Exercise 2b
        # Slicing and printing in one line
        print(warning_phrase[:11])
```

```
        Do not push
```

```
In [5]: # Worked solution for Week 1, Exercise 2c
        # Slicing differentently for another phrase
        print(warning_phrase[7:])
```

```
        push the button
```

Chapter 6
Conditionals, Loops and Functions

6.1 Learning Outcomes

By the end of week 3, you should be able to:

1. Understand the use of tab indents and colons.
2. Use `if` statements.
3. Write `while` and `for` loops.
4. Use format strings (advanced).
5. Incorporate the `range` function into your loops.
6. Use and define your own **functions**.
7. Understand what `lambda` functions are and when to use them (advanced).
8. Use the dictionary data type.

6.2 Colons and Indentation

Two important aspects of `Python` coding are indentation and colons, these help tell the program which parts of the code belong together when it comes to loops and if statements. These are not common to all programming languages (some use curly braces instead of indentation), but are vital for conditional statements, loops, and function definitions; three of the most important tools in programming.

Anatomy of Colons and Indentation

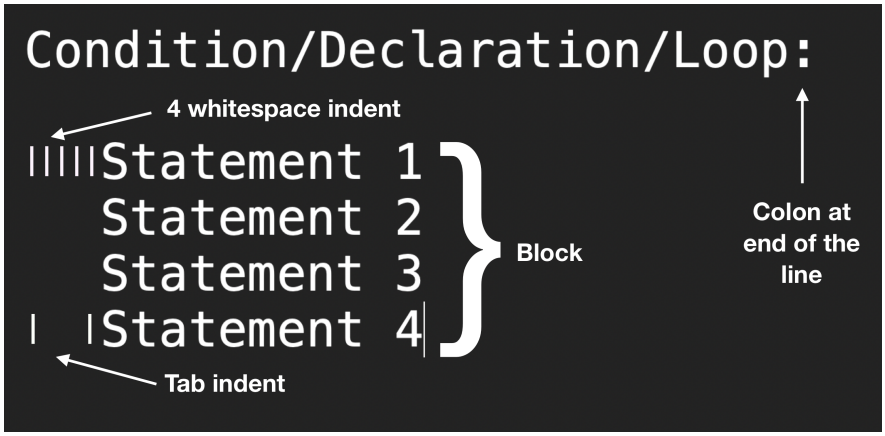


Figure 6.1: A diagram showing the general form of a snippet of `Python` code using colons and indentation. Note: (as mentioned later) you can't mix whitespaces and tab indents, they are purely here for illustration.

Colons are easy to forget, but are straight forward to use. You need to put one at the end of any `Python` command that is declaring a loop, condition (like an if statement), or function (as shown in Fig.6.1). If you forget to do that the code won't run and you'll be given a `SyntaxError` at the location of the line that is lacking one.

Indents are similarly easy to forget, and you'll usually get an error message that tells you where you've forgotten to put one if you run the code. However, when things get a bit more complex and you have multiple nested loops and conditions it can be very easy to mis-indent a block or statement. This can be problematic, either introducing minor undesirable behaviours into your program, or a seriously detrimental bug.
Time for an anecdote: One of the authors of this document managed to make 12TB of data overnight by accidentally indenting a single line of code one step further than it should have been. This caused the University of Sussex's high performance computing (HPC) cluster to fill up overnight, and induced mild panic in the department as everyone tried to clear unnecessary data out before important data was lost. So take this as cautionary tale: **Check your indents**.

An indent is, by convention, 4 "columns" ("white spaces") wide for `Python`, though it is possible to use one tab character instead - **using both in one script will not work**. In general Jupyter will add them for you if you've put a colon at the end of the previous line and then hit ⏎. If you need to put them in yourself, we recommend using the tab key (although that does not work in all editors), rather than counting the number of times you've hit the space bar (which becomes extremely tedious).

Once a line, or series of lines, of code is indented, it is known as a ***block***.

Learn to love indented code blocks: they are ubiquitous, powerful, and make the code more readable and easier to follow. Easily readable code is one of `Python's` main strengths and if you're ever handed a large program someone else has written you will truly begin to appreciate why this matters.

6.3 The if statement

The `if` statement is a conditional construct. If a stated condition is met (is True), the code then executes a command or series of commands in the indented block following the colon. If statements are very useful and are used all the time in real codes.
The if statement allows a block of code run only if a condition is met. There are three types of if statements, **if**, **elif** and **else**. The '`elif`' statement is short for 'else if' and allows for another conditional statement to be added to an existing if statement. If the 'if' condition is not met, then the elif condition is tested, if this '`elif`' condition is met then the 'elif' block (statements indented below elif condition) is executed. If the condition of '`elif`' is not true then that code block is also skipped. The '`else`' statement doesn't use an explicit condition (it has an implied condition simply meaning "If none of the above was True"). For a visual interpretation of this see Fig.6.2.

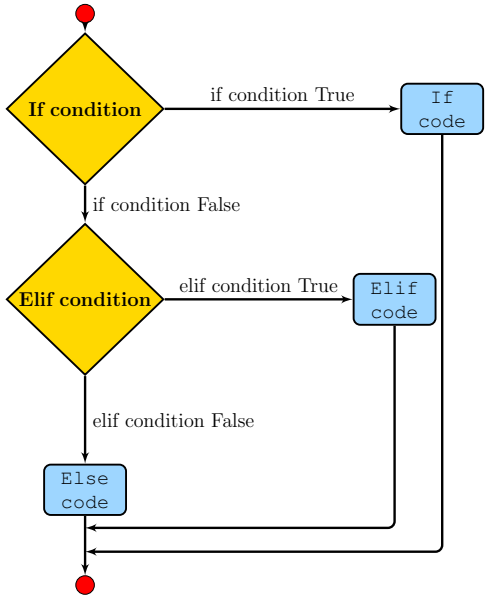


Figure 6.2: A generic "if" statement flow chart

Note: you can have as many '`elif`' statements in an '`if`' construct as you like, but only one '`else`' (this should make sense if you think about it). Neither statement is required though and each use case will define how the conditions should be approached.

Also note: unlike in some programming languages, in `Python` you do not need to spell out where the `if` condition stops, i.e. there is no `endif` statement. This is because the condition only applies to the indented lines of code.

Here is a simple example of how to use an if statement (notice the structure of colons and indentation above (see Section 6.2)):

```
In [2]: # Define a list of integers
        list1 = [0, 1, 2, 4, 5]

        # Test if the list contains 1
        if 1 in list1:
            print('There is a one in list1')
        else:
            print('There is not a one in list1')
```

Now run the cell, and you should receive the following output.

```
        There is a one in list1
```

Now edit the cell and change `list1` to the following.

```
        list1 = [0, 10, 2, 4, 5]
```

Running the code again then gives.

```
        There is not a one in list1
```

As you will see if you have done the task correctly, the '`if`' condition was satisfied in the first case. In the second case it was not, hence the '`else`' block was executed.

Here is another example, this time using an '`elif`' statement (at this point you might want to revise Boolean statements from session 1). Note that this example introduces a way to do nothing in a code block, `pass`.

```
In [3]: # Define a variable containing an integer
        a = 50

        # Test the value of the variable a
        if a == 50:
            print( 'Variable a is 50' )
        elif a > 100:
            print( 'Variable a is more than 100' )
        # Note that the next two lines do nothing, and can be omitted
        else:
            pass  # do nothing

        print ('Finished')
```

Running the code should produce the following result.

```
        Variable a is 50
        Finished
```

Now change a in the code to the following.

```
        a = 120
```

Running it a second time should produce the following result.

```
Variable a is more than 100
Finished
```

As you can see the 'elif' condition was met and block executed.
Now change variable a to the following.

```
a = 100
```

Now when running the code the following should appear.

```
Finished
```

Now that the variable a is neither 50 or greater than 100, the 'else' condition is executed. In this case the code is simply pass which does nothing and continues with the remaining code. Pass is rarely advised(!) but if you were to leave this blank then an indentation error would occur. Try the example below and see what happens when you run it.

```
In [4]: # Define a variable containing an integer
        a = 10

        # Test the value of the variable a
        if a == 50:
            print('Variable a is 50')
        elif a > 100:
            print('Variable a is more than 100')

        print('Finished')
```

The example above should show you that you don't **have** to include 'else' statements with an 'if' statement, often there is no need for one.
You don't just have to use singular conditions for 'if' and 'elif' statements, you can combine conditions together using and or or any other Boolean combination, as in the following:

```
In [5]: # Define a tuple of names
        b = ("Neil", "Edwin", "Michael")

        # Test if certain names are within the tuple
        if "Edwin" or "Buzz" in b:
            print("Edwin 'Buzz' Aldrin was the second man on the moon.")
        # Note that the next two lines do nothing, and can be omitted
        else:
            pass
```

If the code above was run, the text would be printed if tuple b contained 'Edwin' or 'Buzz'. The Boolean not operator could be used to reverse the output i.e True becomes False and vice-verse. For example the above code could be changed to:

```
In [6]: # Define a tuple of names
    # Test if certain names are within the tuple
    if "Edwin" or "Buzz" not in b:
        print("One of his names is missing from the list!")
    # Note that the next two lines do nothing, and can be omitted
```

```
    else:
        None
```

```
In [7]: # Define a tuple of names
    b = ('Neil', 'Edwin', 'Michael')
    # Test if certain names are within the tuple
    if 'Neil' and 'Edwin' in b:
        print('Neil Armstrong and Edwin \'Buzz\' Aldrin were the first
              men to set foot on the moon.')
    # Note that the next two lines do nothing, and can be omitted
    else:
        None
```

This time, the use of and prints the text since both conditions have been met. Here, we have used \' to allow a quotation mark inside a string. The backslash tells Python to ignore parts of its own syntax and just use it as text within the string. Alternatively, you can define the string with double quotation marks to use a single quotation mark in the string, and vice versa.
Additionally you can combine (or "nest") 'if' statements for example below:

```
In [8]: # Define a list of numbers
        q = [2, 4, 6, 8]

        # Test the contents of the list with nested if statements
        if 2 in q:
            if 4 in q:
                if 6 in q:
                    if 8 in q:
                        print ('2, 4, 6 and 8 are in q')
                    else:
                        print ('2, 4 and 6 are in q')
                else:
                    print ('2 and 4 are in q')
            else:
                print ('2 is in q')
        # Note that the next two lines do nothing, and can be omitted
        else:
            pass
```

The indentations also help you see what is happening when using multiple 'if' statements, (i.e. they make the code more 'readable'), since each corresponding pair of 'if' and 'else' statements line up.

6.3.1 Exercises

Exercises (with worked answers)

Try the following examples. (The solutions can be found in section 6.13).

1. Create a list of names called nlist, then create a string variable called myname. Use an if statement to check if myname is not in nlist. If myname is not in the list, add the name to the list and print the list. If it is within the list, print the list. (At this point, you might need to revise the append command from session 2.

2. (a) Set a variable called `myscore` and assign an integer of between 0 and 100 to it. Then using `if` statements print ('You have a first') if `myscore` is 70 or above, 'You have a 2:1' if `myscore` is between 60 and 69, 'You have a 2:2' if `myscore` is between 50 and 59, 'You have a third' if `myscore` is between 40 and 49 and 'You have a not passed' if `myscore` is below 40.

   (b) Adapt your code so that if `myscore` is above 100 or below 0, it replies that '`myscore` is not within the correct boundaries'.

   (c) Adapt the code so that it asks for, and accepts, a value from the user. This value can be an integer or float.

Exercises (other)

1. W3Basic1 - A leap year is defined as a year which is:

   - Divisible by 4. **and**
   - **Not** divisible by 100. **Unless**
   - It is divisible 400.

   Using a single Jupyter cell, write a section of code that allows the user to input a year, then outputs 'True' if the year is a leap year or 'False' if it is not. [Hint: Make use of the modulo operator %, if you have not come across modulo arithmetic the Wikipedia page is pretty good!]

2. W3Basic2 - Write some `Python` commands in a Jupyter cell to check if an input number is positive, negative or zero.

6.4 While Loops

When we want to perform a repetitive task in computing we call upon loops. Using loops incorrectly in your code can lead to "infinite loops", so please have a look at section 4.5.2 to remind yourself what to do if that happens.

There are two basic types of loop: `while` loops and a `for` loops. A while loop executes a piece of code while a condition is considered true.

Until the '`while`' condition is declared false, the code continues to execute the code block inside the `while` loop. In some cases a counter is used to make sure the condition will be declared false at some point (if it is never declared false, the code enters an infinite loop). In some cases an '`else`' statement is used to invoke another code block, i.e. when the opposite of the `while` condition is true. A basic flow chart in figure 6.3 shows a while loop. Note the need for colons after the condition, and indented executable code (as was the case with the `if` statement).

An example of a while loop can be seen below:

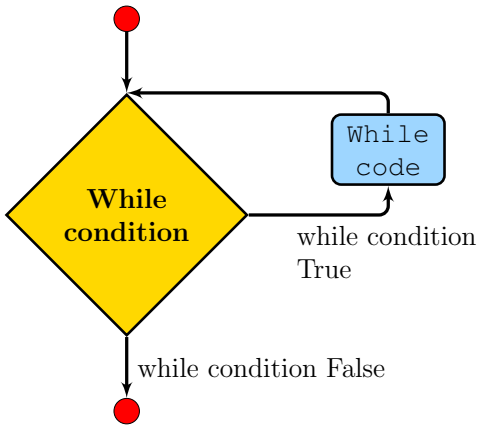


Figure 6.3: While loop flow chart

```
In [1]: while condition:
            # code block 1
        else:
            # code block 2
```

Notice that you can also follow a while loop with an if statement. This essentially says "while the condition is true: execute the while block, once it is not true: execute the else block".

Try to work out what these while loops do and add appropriate comments:

```
In [2]: x = 0
        while x <= 12:
            print(x)
            x = x + 1

        print('Done')
```

The example below contains an 'if' statement nested in a while loop.

```
In [3]: i = 1
        while i < 10:
            if i%2 == 0:
                print (i, ' is an even number')
            else:
                print (i, 'is an odd number')
            i = i + 1
```

In the example above `i = i +1` is used, this takes the variable `i` and adds one to it, an alternative syntax is:

```
In [4]: i = 1
        i +=1
        i
```

```
Out[4]: 2
```

The use of `i += n`, takes variable `i` and adds `n` to it and reassigns it to variable `i`. The examples below show how to similarly take a variable, and add, subtract, multiply or divide, then reassign the new value to that variable. These commands can be useful for counters.

```
In [5]: i -= 1
        i
```

```
Out[5]: 1
```

```
In [6]: i *= 5
        i
```

```
Out[6]: 5
```

```
In [7]: i /= 2.0
        i
```

```
Out[7]: 2.5
```

6.4.1 Exercises

1. W3Basic3 - Write some code to do the following:
   (a) Ask a user to enter a start number, end number, and interval (i.e. step size).
   (b) Then print out a series starting and ending at the defined values in steps of the desired interval.

6.5 For Loops

For loops are similar to while loops but they work through items in a sequence, executing a block of code each time. You will be using them a lot in this module.
If you have not read the while loop section yet, please do so, as this section won't make sense otherwise!

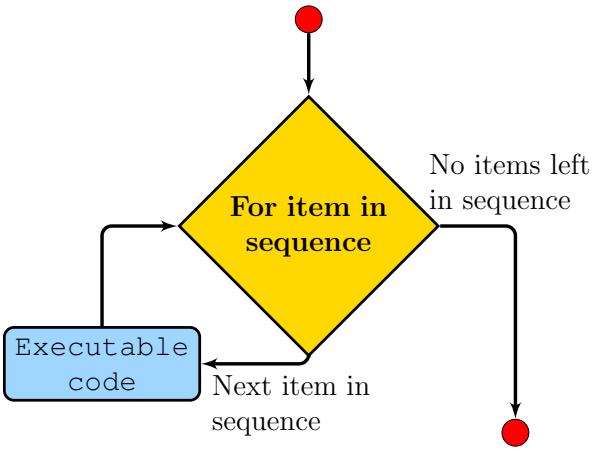


Figure 6.4: For loop flow chart

One way to think of a for loop is that in the background a counter is active, performing a task until the counter reaches a certain number that is equal to the total items in a sequence. Just like `if` statements and while loops, for loops are written in the same way with a block, indented in to tell `Python` what to do.

Similar to `'while'` loops 'else' statements can be used to perform an operation after the loop is complete. `'for'` loops can also be nested within each other to perform operations. An example layout of a for loop can be found below.

```
In [1]: for item in sequence:
                # code block 1
        else:
                # code block 2
```

The sequence can be a range of numbers, a list, tuple, or even a string. The item is commonly referred to as '`i`' as it is a counter starting at the beginning of the sequence and stopping at the end, however we can name this counter variable anything within reason. (Because `i` is so commonly used as a counter, `Python` uses `j` to denote complex numbers, see session 1).

This example prints out the result of a simple maths operation on a list of numbers.

```
In [2]: # Define a list of numbers
        numberlist = [1, 2, 3, 4, 5, 6]

        # Loop over the elements in the list squaring them
        for i in numberlist:
            print (i*i)
```

This example prints each letter of a word on a new line. Here the sequence is a string, i.e. 'word'.

```
In [3]: # Loop over the characters in a string printing each
        for letter in 'word':
            print (letter)
```

This example loops through a list of strings, printing the string and then it's length. Here the sequence is a list, i.e. 'list1'.

```
In [4]: # Define a list of strings
        list1 = ['Monty', 'Python', 'Spam', 'Eggs']

        # Loop over the list printing the word and it's length
        for word in list1:
            print (word, 'has', len(word), 'letters')
```

6.6 Format strings

In the above example you will notice that when printing out the strings and integers or floats etc., parentheses and quotation marks are also printed. This is due to the fact we are mixing data types when printing, a way around this is to assign non-string types to variables and then use *format strings* to print them.

A *format string* is preceded by the letter f, and the variables to be printed are enclosed in curly braces {}, e.g.

```
In [1]: # Example of format strings
# Define a list of strings
        list1 = ['Monty', 'Python', 'Spam', 'Eggs']

        # Loop over the elements in the list
        for word in list1:

            # Get word length
            wl = len(word)
            print(f'{word} has {wl} letters')
```

You can optionally specfify the format of the string conversion by following the variable name with a format qualifier, such as ':3d' to print a 3-digit integer, or ':5.4f' for a float with format x.xxxx.

```
In [1]: # Example of format strings with optional format specifier
print(f'root 2 to 6 sig figs = {2**0.5:6.5f}')
```

6.7 Range command

The `range` command allows you to create a list of numbers. You will be using it a lot in this module. Below are a few examples of the range command in action:

```
In [1]: # Define a list using the range command containing
        # integers from 0 to 9 (inclusive)
        list1 = range(10)
        print(*list1)
```

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

This is the range command in its simplest form, `range(j)`, it creates a list starting at zero with increments of 1 going all the way up to `j-1`. Note the `*`, this expands all the elements of a list (or tuple) rather than printing the list itself.

```
In [2]: # Define a new list using range with a start and end point
        list2 = range(1, 11)
        print(*list2)
```

```
        1 2 3 4 5 6 7 8 9 10
```

The above example shows the `range` command with start and end declarations, i.e. the command `range(i,j)` creates a list starting at `i` and ending at `j-1` with increments of 1.

```
In [3]: # Define a list using range with a start, end and step size
        list3 = range(0, 10, 2)
        print(*list3)
```

```
        0 2 4 6 8
```

This final example shows the `range` command with all of its inputs. `range(i,j,k)` where `i` is the start number, `j` declares the end (`j-1`) and `k` shows the step size. All of the inputs need to be integers as the range command only works with integers and not floats.
Table 6.1 shows all the variants for the range command.

Note: range command only works with integers	
`range(j)`	Creates a list starting at 0 and ending at `j-1` with increments of 1
`range(i,j)`	Creates a list starting at `i` and ending at `j-1` with increments of 1
`range(i,j,k)`	Creates a list starting at `i` and ending at `j-1` with increments of `k`

Table 6.1: Range command

Below is an example using the `range` command and nested `for` loops.

```
In [4]: # Loop through a range
        for i in range(2,6):

            print (f'{i} times table')

            # Within the outer loop, loop over numbers between 1-10
            for j in range(1,11):

                # Compute product of outer loop and inner loop integers
                num = i*j
                print (f'{i} x {j} = {num}')
```

6.7.1 Using the separator command

The separator command in `print` is a nice feature.

```
In [5]: # Define a list from a range
        list4 = range(1, 11)

        # Print the list separated by whitespaces and commas
        print(*list4, sep=' , ')
```

```
        1 , 2 , 3 , 4 , 5 , 6 , 7 , 8 , 9 , 10
```

The separator can indeed be anything, including ridiculous things.

```
In [6]: print(*list4, sep='  [banana] ')
```

```
        1 [banana] 2 [banana] 3 [banana] 4 [banana] 5 [banana] 6 [banana] 7 [
    ↪ banana] 8 [banana] 9 [banana] 10
```

6.7.2 Exercises

Exercises (with worked answers)

Try the following examples. (The solutions can be found in section 6.13).

1. Use a `for` loop and the `range` command to find the first 10 terms of the following series:
$$\frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \frac{1}{16} + \ldots$$

2. Using range, a for loop and if, elif and else conditions, print: `'1 potato 2 potato 3 potato 4, 5 potato 6 potato 7 potato more.'`.

3. Write an program that asks for and inputs the user's name. It then checks if the users name is within a list of names. If the name is in the list, it should then print a message telling the user that their name is on the list. If not, add the users name to the list and inform them their name has been added to the list.

Exercises (other)

W3Basic4 - Write a program using a for loop that allows the user to input a word then print out the word backwards.

6.8 Functions

There are already functions built into `Python`, `print` for instance. However, you can also define your own functions. Functions are repeatable blocks of code that can be used over and over again.

Functions are defined using colons and indentation, similar to `if` statements and for/while loops. You can define parameters (arguments) to be used within the function, a general example is below:

```
In [1]: def function_name( parameter1, parameter2,....):
            # block of code
            return statement(s)
```

Then to call upon them in the script use the following command.

```
In [2]: function_name(x_parameter1, x_parameter2, ...)
```

Arguments don't have to be used, however they are used when performing tasks on different data points etc. An example not containing arguments is below:

```
In [3]: def GameOver():
            print ('Game Over!')
            return

        GameOver()
```

Running the above code, will simply print out `Game Over!` when called upon, this simple example shows how repeatable code can save time as you don't have to write out the print command over and over again.

Be careful not to call your new function something that is already built in to `Python`! You can check whether a function name is already in use by typing `help(NameToCheck)`. As your coding gets more advanced, you will feel the benefits of using functions more and more.

The following example uses arguments:

```
In [4]: def cubed(alist):

            # Loop over elements of the argument
            for i in range(len(alist)):

                # Cube the element and reassign it to the list
                alist[i] = alist[i]**3

        # Define a list of numbers to be used in the function
```

```
        numbers = list(range(5))

        # Cube the numbers using the user defined function
        cubed(numbers)
        print(numbers)
```

Running the code would output, `[0, 1, 8, 27, 64]`. Here, we have passed in the `numbers` variable, which was then assigned to a variable called `alist` in the function which is entirely internal to the function.

```
In [5]: def series(para):

            # Define a variable to hold the sum
            asum = 0

            # Loop over elements of the argument
            for i in para:

                # Sum the squares of the argument
                asum += para[i]**2

        # Define a list of numbers to be used in the function
        numbers = list(range(5))
        # Find the sum of the series defined by the function
        series(numbers)
        print(asum)
```

This code finds the sum of the series x^2, however if you have copied the code and run it yourself, you will find a NameError has occurred. This is due to `asum` not being a global variable i.e. it is only a variable within the function. To use this variable in different parts of the code we will need to return it, which will pass it back out of the function and assign it to another variable (this is part of a greater concept in coding called 'Namespaces' [scopes in other languages] which will be covered later if people are interested).

```
In [6]: def series(para):

            # Define a variable to hold the sum
            asum = 0

            # Loop over elements of the argument
            for i in para:

                # Sum the squares of the argument
                asum += para[i]**2

            return asum

        # Define a list of numbers to be used in the function
        numbers = list(range(5))

        # Find the sum of the series defined by the function
        series_sum = series(numbers)
        print (series_sum)
```

The return statement outputs the value, then we need to assign it to a variable, in this case `series_sum`, to then read from it. If we wanted to return multiple values we can do the following:

```
In [7]: def timestable(number):

            # Compute the products of the number in the argument
            a = number * 2
            b = number * 3
            c = number * 4
            d = number * 5

            return a, b, c, d

        # Compute 5 multiplied by 2-5 (inclusive) using the function
        q, w, e, r = timestable(5)
```

Using commas we can assign multiple variables with multiple returns. Side note here, a function returning multiple variables actually returns a tuple containing the returns.

6.8.1 Exercises

1. W3Basic5 - Repeat W3Basic1 using a function.
2. W3Basic6 - Repeat W3Basic2 using a function.
3. W3Basic7 - Write a new function.
   (a) It must take two parameters, a pay rate, and number of hours worked.
   (b) Make it return the total pay.
   (c) Alter the function so that any hours over 40 is paid at 1.5 times the normal rate.

6.9 Signpost

You need to have at least got to this point by the end of the lab session in week 3. You now have all the tools needed to finish the first assessment.

6.10 Lambda statement

The lambda statement is basically a single line function for use with expressions. The function can be expressed in the following way, with an arbitrary number of arguments:

```
In [1]: function_name = lambda argument1, argument2, ...: expression
```

An example of a lambda statement:

```
In [2]: root = lambda x, n: x**(1 / n)

        print (root(2,2))
        print (root(2,3))
        print (root(2,4))
```

Evidently, they are very similar to the function command but can be more succinct for simple repeatable calculations.

6.11 Dictionaries

Although not used heavily in this course, dictionaries are one of the most powerful data structures in `Python`.
The dictionary data type holds pairs of **keys** and **values**, and are defined using curly braces '{ }'. The key, value pairs are defined as items with commas and a key separated by a ':'.

A dictionary cannot have any repeating keys, however it can have repeating values i.e. different keys can have the same value. An example of how to create a dictionary can be seen below, it also shows how to retrieve information from a particular key. Note that the values do not need to be strings, they can be numbers (e.g. phone numbers), or even `Python` objects.

```
In [1]: # Define a dictionary
        dictionary = {'key':'value', 'another key':'another value'}

        # Extract a value
        dictionary['key']
```

```
Out[1]: 'value'
```

```
In [2]: # Extract another value
        dictionary['another key']
```

```
Out[2]: 'another value'
```

```
In [3]: # Define a dictionary with 2 of the same key
        double = {'key':1, 'key':2}
        double
```

```
Out[3]: {'key': 2}
```

From the example above, you can see that if there is a repeating key the last value will be taken. Below are some more examples of manipulating dictionaries. Here we reassign a string to an existing key overwriting the old value.

```
In [1]: # Define a new dictionary
        dictionary = {'key':'old','Hello':'World'}

        # Overwrite the value of 'key'
        dictionary['key'] = 'new'
        dictionary['key']
```

```
Out[1]: 'new'
```

We can delete a key using `Python`'s `del` function. This function deletes what follows it from memory (technically only removes the label/memory address) and is not specific to dictionaries. If you have a list that is hogging memory, but you no longer need it you can free up that memory using the `del` function (sort of: memory management is a somewhat mystic art in `Python`, ask if you are interested).

```
In [2]: # Delete the entry under the key 'Hello'
        del dictionary['Hello']
        dictionary
```

```
Out[5]: {'key': 'new'}
```

You can add new keys:

```
In [6]: # Assign a new entry to the dictionary
        dictionary['Good'] = 'bye'
        dictionary
```

```
Out[6]: {'Good': 'bye', 'key': 'new'}
```

You can get the length of the dictionary which corresponds to the number of key, value pairs.

```
In [7]: # Get the length of the dictionary
        len(dictionary)
```

```
Out[7]: 2
```

You can also wipe a dictionary completely without deleting the data structure itself thusly:

```
In [8]: # Wipe the contents of the dictionary
        dictionary.clear()
```

```
Out[8]: {}
```

Below is a set of examples showing how you can access specific parts of a dictionary and interact with them. We can get a list containing the items.

```
In [9]: # Define 2 dictionaries for manipulation
        dict1 = {'Hello':'World','Good':'Bye'}
        dict2 = {'Game':'Over','You':'Lose'}

        # Get the items in a dictionary
        list(dict1.items())
```

```
Out[9]: [('Hello','World'),('Good','Bye')]
```

We can get a list containing the keys.

```
In [10]: # Get a list of the keys of a dictionary
         list(dict2.keys())
```

```
Out[10]: ['Game','You']
```

We can get a list containing the values.

```
In [11]: # Get a list of the values of a dictionary
         list(dict1.values())
```

```
Out[11]]: ['World','Bye']
```

We can test if a key is within a dictionary.

```
In [12]: # Test if 'Hello' is in the first dictionary
         if 'Hello' in dict1
```

```
Out[12]: True
```

```
In [13]: # Test if 'Hello' is in the second dictionary
         if 'Hello' in dict2
```

```
Out[13]: False
```

We can combine dictionaries with the update method.

```
In [14]: # Combine the 2 dictionaries
         dict1.update(dict2)
         dict1
```

```
Out[14]: {'Hello':'World','Good':'Bye','Game':'Over','You':'Lose'}
```

The `dict.items()` function lists all the keys and values as tuples. The `dict.keys()` and `dict.values()` functions give the keys and values of the dictionary respectively. Finally the `dict.update()` adds the keys and values from one dictionary to another. - note that the "()" is important - the `.key` etc. operations will not do anything without them. Think back to the function we defined with no arguments, the same would be true there.

6.11.1 A Note On Iterators

Here we have taken a small amount of poetic license. As of `Python` 3 a number of `Python`'s inbuilt functions were changed to return an object called an iterator rather than lists or tuples. This can greatly increase performance when simply looping through a set of keys, values or items in a dictionary but can also lead to some confusion when working with these. This is why all calls to `.items`, `.values` and `.keys` above are wrapped with `list()`.
You can work with the elements inside an iterator without issue when looping through them but should you find yourself needing to extract all values into a list and print them you would find that a `Python` object were printed instead. This is shown below

```
In [1]: # Define a dictionary containing arbitrary keys and values
        dict1 = {'a': 1, 'b': 2, 'c': 3}
        print(dict1.keys())
        print(dict1.values())
        print(dict1.items())
```

```
        dict_keys(['a', 'b', 'c'])
        dict_values([1, 2, 3])
        dict_items([('a', 1), ('b', 2), ('c', 3)])
```

To avoid this you can simply convert to a list or indeed a tuple as shown above depending on whether they need to be mutable or not.

6.12 Advanced Exercises

Helpful hint: checking the glossary and the world wide web, may help with these exercises.

1. Write a program where a user gets 3 chances to guess a password. If the password is correct print 'Access Granted' and exit the loop, if incorrect print 'Access Denied'. Modify the code so that after the third guess a message is displayed telling the user they have run out of guesses.

2. DNA sequence strings are made up of the letters A, T, G, and C. The DNA sequence strings have a complement string where the the letters are switched. A's become T's, T's become A's, G's become C's, and C's become G's. For example, the complement of TTATGGCGTA is AATACCGCAT. Write a function that produces a complement of a DNA string.

3. Using a dictionary create a phone book, with a key of names and numbers as values. Then write a function that searches the dictionary for a name, if the name exists, give options to call, edit or remove the contact. If call is selected, print `Calling`, and the name, to the screen. Whereas, if edit of remove are selected, edit the number or remove the contact respectively. If the name does not appear give the option to add the name to the dictionary.

6.13 Worked Examples

```
# 3.3.1
# Question 1.
myname = 'Nick' # Your name goes here
nlist = ['Amy', 'Tom', 'Nina', 'James']
if myname not in nlist:
        nlist.append(myname)
        print (nlist)
else:
        print (nlist)

# Question 2a.
myscore = 65 # Your score would go here
if myscore > 70:
        print ('You have a first')
elif 60 <= myscore <= 69:
        print ('You have a two:one')
elif 50 <= myscore <= 59:
        print ('You have a two:two')
elif 40 <= myscore <= 45:
        print ('You have a third')
elif myscore < 40:
        print ('You have not passed')

# Question 2b.
myscore = 65 # Your score would go here
if 70 <= myscore <= 100:
        print ('You have a first')
elif 60 <= myscore <= 69:
        print ('You have a two:one')
elif 50 <= myscore <= 59:
        print ('You have a two:two')
elif 40 <= myscore <= 45:
        print ('You have a third')
elif 0 <= myscore <= 40:
        print ('You have not passed')
else:
        print ('myscore is not within the boundary of 0 and 100')
```

```
# Question 2c.
myscore = float(input('What is your score? '))
if 70 <= myscore <= 100:
        print ('You have a first')
elif 60 <= myscore <= 69:
        print ('You have a two:one')
elif 50 <= myscore <= 59:
        print ('You have a two:two')
elif 40 <= myscore <= 45:
        print ('You have a third')
elif 0 <= myscore <= 40:
        print ('You have not passed')
else:
        print ('myscore is not within the boundary of 0 and 100')

3.6.2
# Question 1
series = 0
for i in range (1,11):
        series += ( 1/(2.0**i) )
        print (series)

# Question 2
string = ''
for num in range(1, 8):

    if num in [1, 2, 3, 5, 6]:
        string += str(num) + ' potato '
    elif num == 4:
        string += '4, '
    else:
        string += '7 potato more.'

print(string)

# Question 3.
names = ['Tom', 'Jerry', 'Sarah', 'Julian']
uname = input('Name? :')
if uname in names:
        print ('Your name is on the list')
else:
        name.append(uname)
        print ('Your name has been added to the list')
```

Chapter 7
Python Modules and an Introduction to Plotting

7.1 Learning Outcomes

By the end of week 5, you should be able to:

1. Know what Python modules are and what they are for.
2. Know how to load Python modules.
3. Know how to find out what functions are inside a given module and how to get help with those functions.
4. Be familiar with `numpy`, `scipy` and `matplotlib` modules (especially `arange` from the `numpy` module).
5. Be familiar with the the `random` module.
6. Be able to make basic plots.

7.2 What are modules?

Modules are pieces of code that have been created by other developers that give Python more functions and abilities. They are generally open source, actively developed, and available for free. In serious programs they are indispensable! Replicating code that other people have already written is a pointless waste of time and money (if you're being employed). Often they will have been implemented in a far more memory and time efficient time way than would be possible without years of experience, possibly even interfacing with much faster C or C++ (numpy for instance) code under the hood.
A quick overview of modules:

- Standard Python contains many built in functions, but does not contain everything you will need in your coding future.
- Modules can be added to Python and contain dozens of functions.

- Occasionally (but rarely, this is mostly avoided) function names are duplicated in different modules or between standard Python and a module. Even though they share a name the operation of these functions could be very different. Therefore, you have to be careful to use the correct version (by specifying the module name explicitly in the function call).
- You should only import the modules that you need, and only import them once (loading modules multiple times and/or loading modules you don't need leads to wasted RAM).

Terminology gets a bit muddy from here on out... strictly a function is a self defined chunk of reusable code, as you have seen in the previous chapter. These modules in fact contain "methods", a term which will make far more sense if you progress to object-orientated `Python`. For simplicity we will use the two interchangeably, mostly focusing on mis-using "function" but don't be confused if you come across "method" in the future or online.

7.3 Common Python modules

Modules we will cover in this course[1] include:

- `matplotlib`: graphics tools to create and display plots. In Physics and Astrophysics it is frequently used to make plots for publications.
- `numpy`: data structures and mathematical resources not included in basic Python, for example cosine, exponentials etc. Some basics of `numpy` will be discussed in this section but chapter 9 will provide a more comprehensive overview.
- `scipy`: scientific resources such as those needed in mathematical physics, e.g. bessel, gamma, beta functions, signal processing, integration, differential equation solving and statistics... (you will use `scipy` a **lot** in Scientific Computing in Y2).
- `random` (Section 7.6): used to generate pseudo-random numbers within the Python environment. The module is very useful for generating random numbers quickly in a range of circumstances.
- `pandas` (see Chapter 8.1): allows you to work with data in something analogous to spreadsheets.
- `os`: allows you to interface Python with Unix, for example if you want to load up certain types of file into your Python code.

[1] We are using "course" rather than "sub-module" to avoid confusion with the topic of this lab session.

Here are some additional modules that you may come across, especially if you are working on the optional challenges or an independent project. Although it is absolutely fine to use these, or other, non-standard modules in your optional work, please do not use them in the assessments.

- `TkInter`: a way to use GUIs with Python.
- `PyGame`: if you are writing games.
- `SQLAlchemy`: if you want to set up a database.
- `SymPy`: Symbolic equation solving in python, this is essentially free Maple.
- `time`: amongst other things, this contains a useful function to check how long your code is taking to run.
- `html`: write webpages in python.
- `astropy`: if you are lucky enough to do any astronomy research, then you'll definitely be using this one. It also contains an invaluable sub-module for working with and converting units, as well as another sub-module containing all the useful constants your heart could possibly desire.
- `scikit learn`: a suite of simple machine learning algorithms.
- `tensorflow`: a more in depth module filled with machine learning "machienery", including lots of GPU optimisation.

You can find many more modules described online, e.g. check out
`https://wiki.python.org/moin/UsefulModules`

7.4 Importing Modules

You should import all modules at the TOP of your scripts: loading them anywhere else can quickly lead to confusion, i.e. DO NOT import modules in any other cell than the first in `jupyter`.

There are several different ways of importing modules. The different methods have advantages and disadvantages based upon the task. Until you get more used to Python, you should keep checking with the ATs to make sure you are using the correct method. For now you'll be using one or more of the following methods in `jupyter`. When you become more proficient, you might well be driving python from the command line, or from inside a .py file, and you can load the modules in the same way for those.

7.4.1 Method One

The simplest way to import a module is the following. This example shows how to import the `numpy` module:

```
import numpy
```

This method allows you to access all the functions in a module, without explicitly importing them into your code and using up your computer's RAM (Random Access Memory- the fast access memory used for running processes on your computer, not used to store data). To use a function from that module within the code, we need to tell python the function is associated with the module by joining the function to the module using a full stop, i.e. `module.function`, e.g.:

```
In [1]: import numpy
        print(numpy.exp(1))
        print(numpy.pi)
```

```
        2.718281828459045
        3.141592653589793
```

7.4.2 Method Two

This method imports the module with a custom name for use when calling upon functions, this saves time with modules with long names etc.

```
In [2]: import numpy as np
        print(np.exp(1))
        print(np.pi)
```

Note that you are not obliged to use *np* as the short version for the `numpy` module. You could use *daffodil* if you wanted, but of course you'd not save any time that way! Also note that you can still use the full name of the module, in addition to the short hand. You will find that there are accepted names for a huge number of commonly used modules such as np:numpy, plt:matplotlib, tf: tensorflow etc.

7.4.3 Method Three

The third method imports all the functions from a module into Python for use.

```
In [3]: from numpy import *
```

This simply tells Python, to import all functions from numpy. To then call upon the function just type the function name:

```
In [4]: print(exp(1))
        print(pi)
```

```
        2.718281828459045
        3.141592653589793
```

The main disadvantages of this method are that you will waste memory (RAM) and, when importing multiple modules, you might be importing similar functions with the same names. You then have to take extra care to make sure you are using the function you intended.

Do not use Method three. It wastes memory, time, and can potentially cause massive conflicts with other modules, if you're importing more than one; it is bad Python practice. Using abrreviations, or more officially: "namespaces", like `np` makes the code easier to read, allows other developers to know where the function comes from, and most importantly only loads it into memory when the function is actually used.

Method three has been described here purely so that you are aware of its potential use, especially when looking online for help.

7.4.4 Method Four

The fourth method is similar to the third method, however this time you can specify which functions from a module you wish to use - this method is often the best method when you only want to use one or two functions from a library.
Please restart your kernel now.

```
In [1]: from numpy import exp,sin
```

You can import as many of the functions as you like from the module all separated by a comma.

```
In [2]: print(exp(1))
        print(sin(3.14)) #note that Python defaults to radians
```

```
        2.718281828459045
        0.0015926529164868282
```

If you try to use a function you did not import a NameError will occur, e.g. try:

```
In [3]:  print(pi)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-15-9e2d2bd32686> in <module>
----> 1 print(pi)

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

7.5 Accessing help with modules and functions

7.5.1 The `dir` command

To find what functions are within a module we can use the `dir` function. For example, the following shows the contents of the `random` module (note that only the last part of the `dir(random)` response is shown):

```
In [4]: import random
        dir(random)
```

```
Out[4]: .....
        'randint',
        'random',
        'randrange',
        'sample',
        'seed',
        'setstate',
        'shuffle',
        'triangular',
        'uniform',
        'vonmisesvariate',
        'weibullvariate']
```

So using the `dir` command enables us to learn about a module.

7.5.2 The `help` command

In a Python command line, typing `help( )` with a function name in between the parentheses will display a help file about that function (if it exists). In the example below, we look for information on the range function (only the first part of the `help(range)` response is shown):

```
In [5]: help(range)
```

```
Out[5]: Help on class range in module builtins:

        class range(object)
         |  range(stop) -> range object
         |  range(start, stop[, step]) -> range object
```

If you mistype a function name, you will get a `NameError`. See what happens when you type `help(renge)`.
The help command will tell you what inputs you have to give to the function, as well as any that are optional. If you see a function with parameters in square brackets, [], then these input are optional. Do not type the function out with the square brackets as you see in the help screen, you'll get a syntax error:

```
In [6]: range([1,]40[,2])
```

```
Out[6]: File "<ipython-input-19-da85a1b3f88c>", line 1
        range([1,]40[,2])
              ^
        SyntaxError: invalid syntax
```

When `help()` is not actually helpful

Python is free, and sometimes the help pages reflect this, i.e. they are not useful, especially for beginners. For example, try `help(sin)` (you need to have loaded it from `numpy` first). If you get stuck and `help()` doesn't help, then don't despair! You can ask your AT, or Google, or StackOverflow, or a friend for help.
A quick sidenote, taking breaks definitely helps when you are coding. Sometimes, you need to go to sleep for the solution to a problem to show itself. Also, make sure you don't suffer alone, complain about your issues to your friends, sometimes talking about the problem can help the solution pop up. In fact, a very helpful process in coding called "rubber ducking" involves chatting rubbish to an inanimate object until the issue resolves itself in your head, feel free to substitute the inanimate object for a weary friend. Sometimes talking through the problem is all you need.

7.5.3 Exercises

Exercises (with worked solutions)

These exercises require the `numpy` module. Please try to figure out which functions to use and how to use them by yourself (make use of the numpy documentation and the `help` function) before checking the answers (section 7.13). Learning how to figure out how to do things in Python that you've not been explicitly taught is an essential skill.

Compute the following:

1. $\sin(\frac{\pi}{3})$
2. $\frac{\log(100)}{\log(10)}$
3. Convert 60 degrees to radians.
4. $\log_2(6)$
5. $\ln(5)$
6. Check if $4 \times \arctan(1)$ is equal to π (print True or False)

Exercises (other)

1. W5Basic1 - Write a function that checks Euler's formula,

$$e^{\pm i\theta} = \cos\theta \pm i\sin\theta,$$

for any argument θ.

(a) Check that the right hand side is equal to the left hand side.

(b) It must return True if the formula holds.

2. W5Basic2 - Evaluate $\log_6(2)$ (hint: use the logarithm base change rule).

7.6 The `random` module

In Physics research and statistics, random numbers are used all the time for many purposes; from making random sub-samples of a large dataset to helping us sample highly complex mathematical functions. In this course we'll use randomly generated numbers from a uniform (all numbers in range equally possible) probability distribution and from a Normal (or Gaussian) probability distribution.

7.6.1 Exercises

Exercises (with worked solutions)

Now it's time for some self-driven learning, using the skills taught above: `dir`, `help` and indeed the internet which no programmer ever works without, try these exercises. (for answers see section 7.13).

1. Investigate the `randint, randrange, random`, `uniform` functions, and `gauss`.
2. Generate a random number.
3. Generate a random number between 1 and 10.
4. Generate a random integer between 1 and 100.
5. Generate 100 random numbers from a normal distribution with $\mu = 5$ and $\sigma = 2$ and store them in a list.

Exercises (other)

1. W5Basic3 - Write a function that displays a random letter from an inputted name.
2. W5Basic4 - Generate a random number between 1 and 100 that is divisible by 9. Hint: use `random.randrange`.

7.6.2 Seeding

Often when working with random numbers we may want random numbers but also repeatable results for debuggig/testing. Now... this seems a little paradoxical but it is a very real requirement which you will undoubtedly encounter at some point. To achieve this we can simply "seed" our random numbers. To do this with `random` we simply employ the seed function:

```
In [7]: random.seed(1)
```

Now our random numbers will be the same sequence regardless of when we run it. The only requirement of seed is that the argument (1 here) is an integer, this integer can be any integer and defines the random sequence deterministically. If no seed is set the random numbers are simply seeded by the current system time in milliseconds (ticks) from 1970.

7.7 The `arange` command

The `arange` function is part of the `numpy` module. You can think of it as an advanced version of the built in `range` function (refer back to section 6.7). Unlike the regular `range` function, the `arange` function allows floats, so you can have step sizes of 0.5 or 0.1, etc. The function doesn't return a vanilla Python list, but instead a numpy array (the difference is akin to that between a vector and a matrix, for more see Chapter 9).

```
In [8]: from numpy import arange
        x = arange(1, 10.5, 0.5)
```

The above generates an array from 1 to 10 in intervals of 0.5. The command is very similar to `range` and the same argument layout can be inferred. Below are some of the outputs using the `arange` function. Try all of these in your Jupyter notebook.

```
In [1]: arange(4.0)
```

```
Out[1]: array([0., 1., 2., 3.])
```

```
In [2]: arange(2, 5)
```

```
Out[2]: array([2, 3, 4])
```

```
In [3]: arange(1, 2, 0.25)
```

```
Out[3]: array([1., 1.25, 1.5 , 1.75])
```

Table 7.1 summarises the options available with the arange function.

`arange(j)`	Creates an array starting at 0 and ending at `j-1` with increments of 1
`arange(i, j)`	Creates an array starting at `i` and ending at `j-1` with increments of 1
`arange(i, j, k)`	Creates an array starting at `i` and ending at the nearest number to `j` based on the increments of `k`

Table 7.1: How to use the `arange` command

7.8 The `matplotlib.pyplot.plot` function

The `matplotlib` module is commonly used to make graphs and figures in Python, it is very easy to use and with some practise can make publication quality plots. Below is an example of how to plot the simplest mathematical equation, $y = x$.

```
In [8]: import matplotlib.pyplot as plt

        # Set up variables for plotting
        x = np.arange(1, 5.5, 0.5)
        y = x

        # Plot the variables
        plt.plot(x, y)
        plt.show()
```

The initial line of the code imports the `pyplot` package from matplotlib. Then the `arange` command creates a range of x values from 1 to 5 in increments of 0.5. The `y=x` line just links the variable name y to the x variable. The `plot` command then generates a line plot. However, this just generates the line on internal matplotlib "figure", to see it we need to `show()` it. This is done with the `show()` command which displays the plot on screen. The output can be seen in figure 7.1.

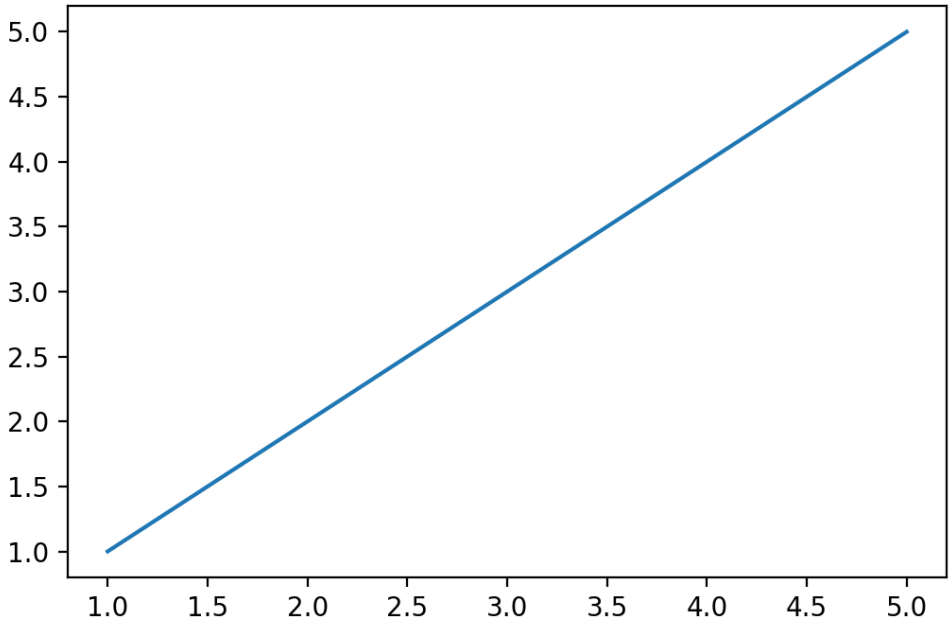


Figure 7.1: Plot of $y = x$.

You may have noticed a little fib here... in `jupyter` the plot is shown regardless of whether `plt.show()` is called. This is for the same reason that the print function isn't required to print variables, thus it isn't true when not using `jupyter`, you should therefore get in the habit of using `plt.show()`... it is more correct after all.

You can also change the limits of the graph. Below is an example of $y = x^2$, with set limits. The output is shown in Figure 7.2.

```
In [9]: # Set up a range of x values
        x = np.arange(-5, 5, 0.01)

        # Compute the square of the x values
        y = x**2

        # Plot the result
        plt.plot(x, y)

        # Set axis limits
        plt.xlim(-1, 2)
        plt.ylim(-1, 5)

        plt.show()
```

`xlim` and `ylim` set the limits, and they need to come after the `plot` command, but before the `show` command, as `plot` draws the plot, then the limits are added before the graph is shown on screen.

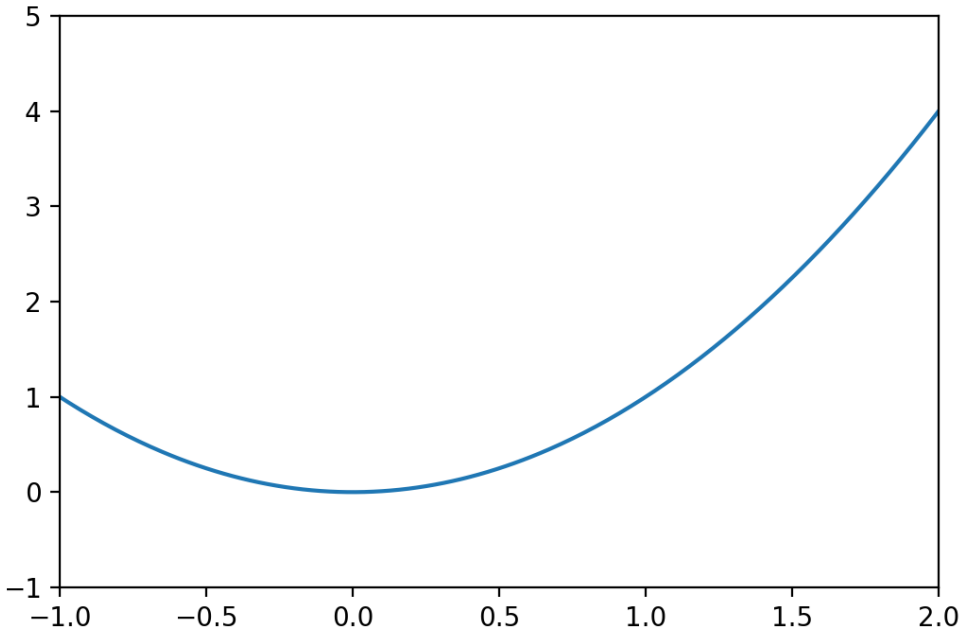


Figure 7.2: Plot of $y = x^2$.

Additionally you can add labels to your axis and a title to your graph. Below is an example of $y = \sin(x)$.

```
In [10]: # Set up a range of x values
         x = np.arange(-20, 20, 0.001)

         # Compute the sin of these values
         y = np.sin(x)

         # Plot the result
         plt.plot(x, y)

         # Set a title and label the axes
         plt.title('Graph of y = sin(x)')
         plt.xlabel('This is the x axis')
         plt.ylabel('This is the y axis')

         # Draw lines to mark 0 on the x and y axes
         plt.axhline(0, color='black', linestyle='--')
         plt.axvline(0, color='black', linestyle='--')

         # Set the y-axis limits
         plt.ylim(-1.2, 1.2)

         plt.show()
```

Just as when adding limits to the plot, any commands to set titles or labels have to come after the initial `plot`. `title`, `xlabel`, and `ylabel` are pretty self explanatory; you have to pass your chosen title/label as a string, which is then added to the plot (these can include LaTeXformatting, i.e. $y = \sin(x)$). The `axhline` (for horizontal lines) and `axvline` (for vertical lines) add `y=0` and `x=0` axis to your graph. Here we have added a dashed line by setting the `linestyle` argument. The plot produced by this code can be seen in Figure 7.3.

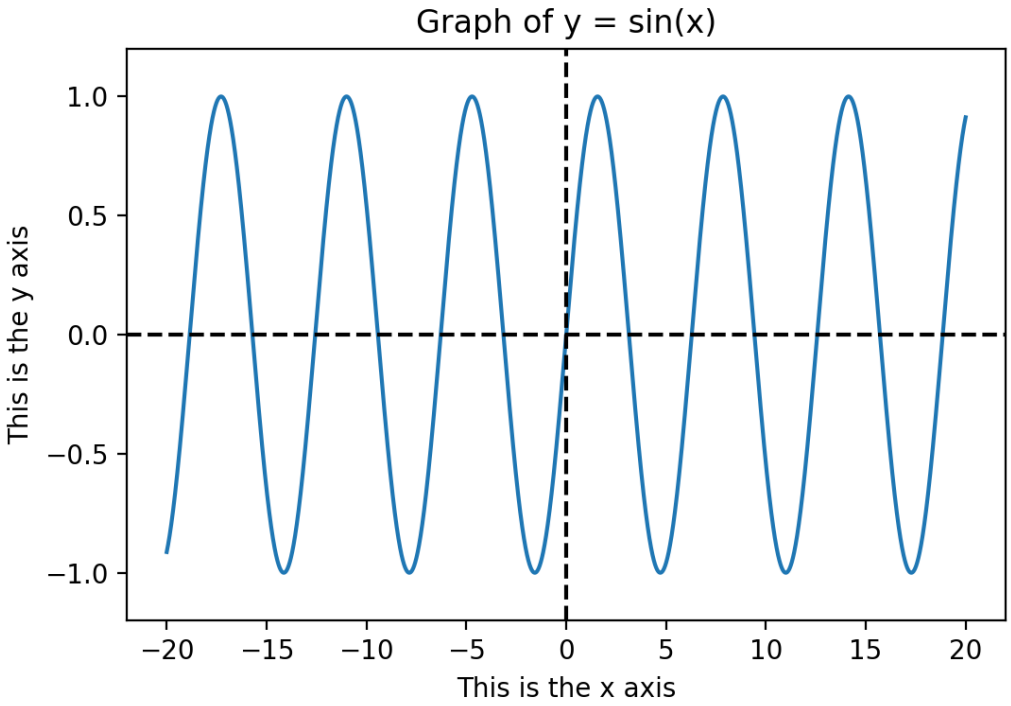


Figure 7.3: Plot of $y = \sin(x)$.

7.8.1 Exercises

1. W5Basic5 - Plot a graph of $\cos(x)$
2. W5Basic6 - Plot a graph of $\cosh(x - 20)$
   (a) Limit the x-axis to between -20 to 5.
   (b) Add an appropriate title, set the fontsize to 14.
3. W5Basic7 - Plot a Gaussian distribution with mean of 0 and a variance of 1.
   (a) Look up the equation for the Gaussian probability density function.
   (b) Generate some x values between -3.2 and 3.2, then use the equation to calculate the equivalent y-values.
   (c) Plot the Gaussian, and add an appropriate title.

7.9 Plotting in multiple figures

When you have more than one graph to display, you have to use the `figure` function. This allows you to separate what you're plotting, otherwise everything will end up on the same set of axes. In the example below we generate two plots and display both below one `jupyter` cell, give it a try.

```
In [11]: # Generate a range of x values, with a small step size
         x = np.arange(-20, 20, 0.001)

         # Create figure with id 1
         plt.figure(1)

         # Calculate the sin of all the x values
         y_sin = np.sin(x)

         # Plot the result in figure 1
         plt.plot(x, y_sin)

         # Add a title to figure 1
         plt.title('Graph of y = sin(x)')

         # Set the y limit for figure 1
         plt.ylim(-1.2, 1.2)

         # Create figure with id 2
         plt.figure(2)

         # Compute the cosine of the x values
         y_cos = np.cos(x)

         # Plot the result in figure 2
         plt.plot(x, y_cos)

         # Add a title to figure 2
         plt.title('Graph of y = cos(x)')

         # Set the y limits of figure 2
         plt.ylim(-1.2, 1.2)

         plt.show()
```

Once you've plotted something, you're also able to close it; this can be particularly useful if you're creating a very large set of plots, as they can start to slow down your computer. To close every open plot, you can use `plt.close("all")`, or if you only wanted to close the first plot from the above example, we could run `plt.close(1)`.

7.10 Plotting a Scatter with `plt.scatter`

Often you will have data that doesn't lend itself to plotting with a nice line. Instead you'll need to plot a scatter of points and maybe plot a theory as a line overlaid over the top. Lets use some of what we learnt with the random module to create a nice messy scatter plot. This can be achieved using `plt.scatter`.

```
In[12]: # Initialise lists to store x and y values
        x = []
        y = []

        # Loop getting 500 random numbers between 1 and 10
        for i in range(500):
            x.append(random.uniform(1, 10))

        # Loop again but this time use a while loop (theres no difference)
        i = 0
        while i < 500:
            y.append(random.uniform(1, 10))
            i +=1

        # Plot the resulting scatter
        plt.scatter(x, y, marker='+')

        # Add a title
        plt.title('My Mess')

        # Label the axes with latex
        plt.xlabel('$x$')
        plt.ylabel('$y$')

        # Add a grid
        plt.grid(True)

        plt.show()
```

Here we have simply extracted 500 random numbers from a uniform distribution and plotted them with "+"s. This marker argument can be a number of things, for a full list look online or in Tab. 8.2, some examples include `'.'`, `','`, `'o'` and `'+'` used here. By default it uses circles (`'o'`). We have also employed the use of LaTeX math mode in the labels and drawn a grid on the background, this can make plots easier to read and is often recommended. The output is shown in Figure 7.4.

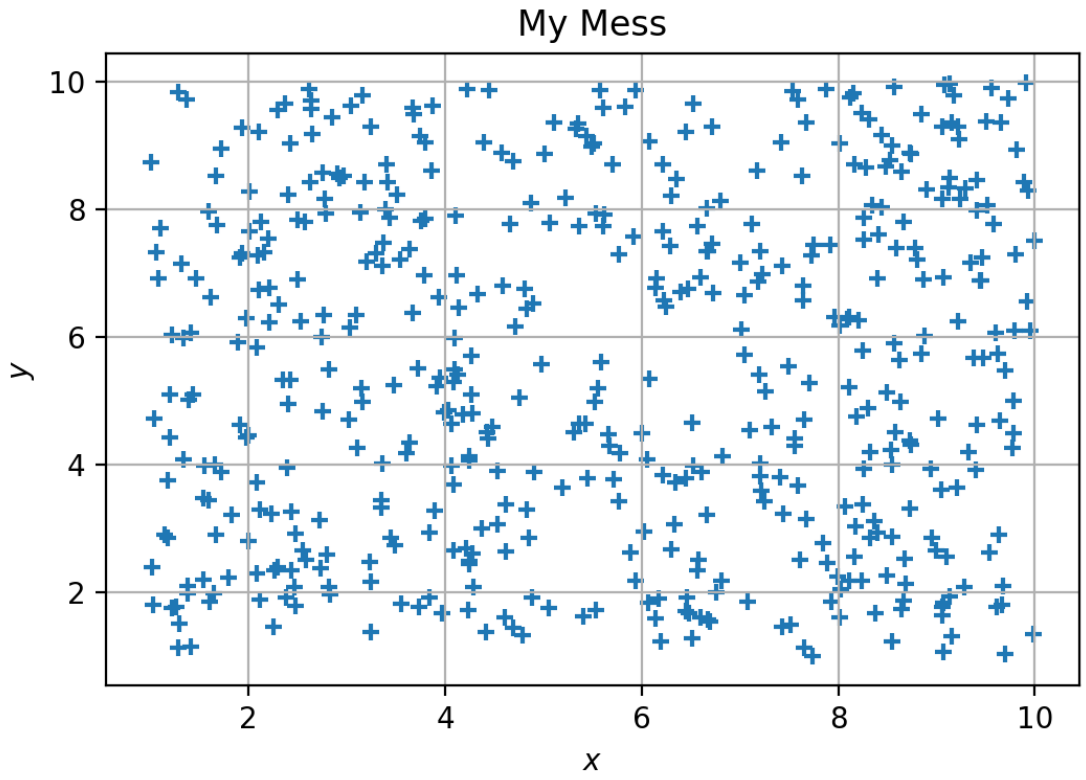


Figure 7.4: An example scatter plot.

7.10.1 Exercises

1. W5Basic8 - Write a code that makes two figures
   (a) Limit both figures to -5 to 5 on the x-axis, and -1.2 to 1.2 on the y-axis.
   (b) Plot $\sin(2x + 1)$ on the first figure.
   (c) Plot $\cos(2x + 1)$ on the second figure.
   (d) Make sure to give both plots appropriate titles, x-axis labels, and y-axis labels.

7.11 Signpost

You should have got to at least this point by the end of Session 5. By now you have the tools to complete parts a) to c) of the second assessment. If you have time at the end of Session 5 we suggest you work through the advances exercises.

7.12 Advanced Exercises

1. Use the random module to create a coin flip simulator. Flip the coin 100 times and print out how many times it landed on heads, and how many times on tails (think about what the results should be before you run it).

2. Plot $\sin(x)$ and $\cos(x)$ on the same graph, with limits -10 to 10 on the x-axis.

3. Plot $\sin^2(x) - \cos(x)$ with limits 1 to 5 on the x-axis and 0.9 to 1.3 on the y-axis. Change the line colour to yellow and line width to 5 (use `help` and/or the matplotlib documentation).

4. Imagine you have some scattered data with a linear relation. Write your own function that naively computes the values `m` and `c` from $y = mx + c$ for a 1 dimensional linear line of best fit.

5. Use the gauss function from the random module to generate 1000 numbers from a Gaussian distribution with a mean of zero and a variance of one, then plot the data as a histogram with one hundred bins. Hint: Use the pyplot.hist command instead of pyplot.plot.

7.13 Worked Examples

These are the solutions for section 7.5.3:

```
In [1]: # Question 1
        print(np.sin(np.pi/3))
```

```
Out[1]: 0.8660...
```

```
In [2]: # Question 2
        print(np.log(100)/np.log(10))
```

```
Out[2]: 2.0
```

```
In [3]: # Question 3
        print(np.radians(60))
```

```
Out[3]: 1.047...
```

```
In [4]: # Question 4
        print(np.log2(6))
```

```
Out[4]: 2.584...
```

```
In [5]: # Question 5
        print(np.log(5))
```

```
Out[5]: 1.609...
```

```
In [6]: # Question 6
        print(4*np.arctan(1) == np.pi)
```

```
Out[6]: True
```

These are the solutions for section 7.6.1:

```
In [1]: # Question 2
        import random
        print(random.random())
```

```
In [2]: # Question 3
        print(random.uniform(1.0,10.0))
```

```
In [3]: # Question 4
        print(random.randint(1,100))
```

```
In [4]: # Question 5
        lst = []
        for ind in range(100):
            lst.append(random.gauss(5,2))
```

Chapter 8
Introduction to Data and Pandas

8.1 Learning Outcomes

By the end of week 6, you should:

1. Be familiar with the `csv` module.
2. Be familiar with the `pandas` module.
3. Be able to perform simple manipulation of pandas databases.
4. Be able to make intermediate level plots (including adding error bars).

8.2 Loading CSV files

You will be familiar with spreadsheets in Excel. These are more generally (outside the world of Microsoft) described as comma-separated values (or csv) files, in fact you may have seen the ability in Excel to export to a csv file. Python has a built in module for accessing the data in csv files. To import the `csv` module just repeat one of the techniques you used in Section 7.4.

Remember that you should always load your modules at the start of a set of code (i.e in the first `Jupyter` cell) and only load a given module once. Note that, if you are using multiple cells and each one has one or more modules loaded, it'd be worth restarting the kernel and clearing output now and again to clear the memory.

You will now need to download the `PythonExampleCSVfile1.csv` file in order to complete the exercises below. The file is on Canvas under Week 6.

If you double click on the icon of this file, it opens in Excel or whatever software is your default spreadsheet software. If you do that, you can see the sort of information it contains. This file was generated by a PhD student, and contains real astronomy research data; however, for these exercises, the meaning of the data is irrelevant.

8.2.1 Specifying Directories

When on university computers, the path to a file in Downloads, Desktop, Documents are, respectively:

```
C://Users/<insert username>/Downloads/<insert filename>
C://Users/<insert username>/Desktop/<insert filename>
C://Users/<insert username>/Documents/<insert filename>
```

For example, if user jb007 wants to load `Python1718exampleCSVfile1.csv` from the Downloads folder, they would do the following

```
In [1]: # Define the file path as a variable
        path="C://Users/jb007/Downloads/PythonExampleCSVfile1.csv"

        # Open the file and do something
        with open(path) as afile:
            #Do something with the file
            pass
```

This will be similar on your personal computers but not exactly the same. Have a look at the filepath displayed for the file, and then copy this into the path variable as shown above. *Within the PIP-Python module, we recommend that you copy any input CSV files to the same directory as your notebook (this can be done using File Explorer on a Windows machine, or Finder on a mac). Doing so means that you do not need to specify the file path in your notebook; see the next subsection for an example. This is particularly important when you submit assignments, as any hard-wired file paths will not work for your marker.*

8.2.2 The `csv` Module

This example assumes the CSV file is in the same directory where you are running Jupyter. If you do not know how to copy the .csv into the same directory as your Jupyter notebook, then ask an AT at the next lab session. In the mean time, refer to Section 8.2.1.

```
In [1]: import csv

        with open('PythonExampleCSVfile1.csv','r') as myfile:

            # Assign the data in the file to a variable
            mydata = csv.reader(myfile)

            # Loop over rows in the dataset
            for myrow in mydata:
                print (myrow)
```

Write these statements into a Jupyter cell to make sure they work for you. If you get an error similar to the one below then the .csv file is not in the right directory (ask an AT for help).

```
IOError: [Errno 2] No such file or directory: 'PythonExampleCSVfile1.csv'
```

Lets go through the above example step by step:

- This method of loading data from CSV files uses code blocks (i.e stuff you would indent).
- The `with` statement is similar to the `if` statement, and so command lines after that need to be indented. Think of the `with` statement as meaning *'with this_item do the following'*. Normally the item being referred to is a file. More detail on `with` will be presented later in the course.
- The `open` command then opens the file, which is defined in a string (or you could use a variable that has been set to the filename).
- The `'r'` statement that comes after the filename will be unfamiliar to you, but is very important. This statement (or equivalent statements) define the *mode* of the file. In this case 'r' stands for read. Again, more on this later.
- We then use '`as myfile`' to assign the file to the variable `myfile`.
- The next line in the code block uses the `csv.reader` function to read in the file to the variable `mydata`. Then a `for` loop is used to print each row of `mydata`, by assigning each row, in turn, to the variable `myrow`. The variable names (`myfile`, `mydata`, `myrow`) are irrelevant, e.g. even if you changed the code to say '`mycolumn`', the data would still be read in as rows.
- After the `for` loop is finished and the with block has been exited, you cannot try to read in the `data` variable, as once the file has been read it is closed. Practice making that error now, because it is more than likely you will make that mistake by accident in future, and its good to get familiar with the error codes, so you can diagnose them quickly.

To be able to use all the data within the CSV file, we have to assign the data to lists. The contents of the file `PythonExampleCSVfile1.csv` can be seen in Table.8.1.

Table 8.1: Contents of PythonExampleCSVfile1.csv

Column A:	name
Column B:	redshift
Column C:	mean-x
Column D:	mean-x minus delta-x
Column E:	mean-x plus delta-x
Column F:	mean-y
Column G:	mean-y minus delta-y
Column H:	mean-y plus delta-y

The use of the `csv` module then requires you to iterate through each row and column and assign the data to lists... This is computationally expensive to do yourself when the files get large (GB), and tedious at best... Fortunately there is a better way to load data into Python!

Variable Names

Please be careful when naming variables for files - if you try to use a variable like `list` or `file`, you will be overwriting a built-in python function. As a general rule, if your variable turns green in `Jupyter` - do not use it!
If you accidentally do this - restart your kernel.

8.2.3 The pandas Module

Pronounced just like the adorable black and white bears, this module is one of the easiest ways to start playing around with data in Python. Lets start by loading the csv into a pandas data frame (think of a data frame exactly like a Sheet in Excel or Google Docs).

```
In [2]: import pandas as pd

        # Open the csv file as a dataframe
        df = pd.read_csv("PythonExampleCSVfile1.csv", index_col=None)
```

The standard abbreviation for the `pandas` is *pd*, similar to how the `numpy` module from last week was `np`.

To read in the csv file, we use the aptly named method `read_csv`. This method can take only one argument, the csv filename, however, we should also tell `pandas` not to read the first column as an index column. Think of the index column as the row numbers in Excel, pandas allows both `int` and `str` indices, but for now we will tell pandas that our file doesn't have an index column already; `pandas` will create one for us using integers from 0 to the number of rows in the csv. (Note: `pandas` assumes this by default, however it is incredibly useful for readability if this argument is explicitly given. This is true of any argument to any function, even if using the default value it can be very helpful for understanding to explictly state the arguments.)

Another bonus of using `pandas` is that it assumes the first row in the csv are the column names - and automatically uses them. Take a look with the `columns` attribute.

```
In [3]: df.columns
```

```
Out[3]: Index(['Name', 'Redshift', 'Mean_X', 'Mean_x_minus_delta',
        'Mean_x_plus_delta','Mean_y', 'Mean_y_minus_delta',
        'Mean_y_plus_delta'], dtype='object')
```

We can also take a look at the data frame using the `head` function to get the first N rows (here 10).

```
In [4]: df.head(10)
```

And because of the way `jupyter` and pandas work well together, you should get a nice table shown that should look something like Figure 8.2.

Out[4]:

	Name	Redshift	Mean_X	Mean_x_minus_delta	Mean_x_plus_delta	Mean_y	Mean_y_minus_delta	Mean_y_plus_delta
0	XMMXCS J113313.8+662243.9	0.12	5.031	4.805	5.255	11.087	10.239	11.883
1	XMMXCS J215336.9+174146.4	0.25	10.599	10.301	10.897	10.363	10.066	10.660
2	XMMXCS J164020.3+464226.2	0.23	10.589	10.134	11.045	10.220	9.875	10.563
3	XMMXCS J172226.8+320754.3	0.23	8.548	7.972	9.125	9.069	8.490	9.695
4	XMMXCS J003706.3+090924.6	0.27	9.089	8.794	9.387	8.693	8.401	8.987
5	XMMXCS J022154.8-054519.0	0.26	8.675	5.200	17.719	8.684	5.200	17.718
6	XMMXCS J233757.0+271121.0	0.12	3.588	3.200	4.025	8.671	8.507	8.835
7	XMMXCS J021837.8-054037.0	0.27	1.244	1.025	1.389	8.536	6.916	10.685
8	XMMXCS J014430.3+021237.0	0.17	3.023	2.279	4.152	8.331	7.272	9.165
9	XMMXCS J131129.9-012024.4	0.18	8.632	8.561	8.703	8.281	8.207	8.356

Figure 8.1: Output of df.head(10) showing the first 10 rows in PythonExampleCSVfile1.csv

If we want to plot just the *Mean_X* and *Mean_y* columns against each other, we can use the column names similarly to how we have previous used functions within modules.

```
In [5]: import matplotlib.pyplot as plt

        # Set up plot
        plt.figure()

        # Plot a scatter of the mean x and y data
        plt.scatter(df.Mean_X, df.Mean_y, marker=".")

        # Label axes
        plt.xlabel("Mean X")
        plt.ylabel("Mean Y")

        plt.show()
```

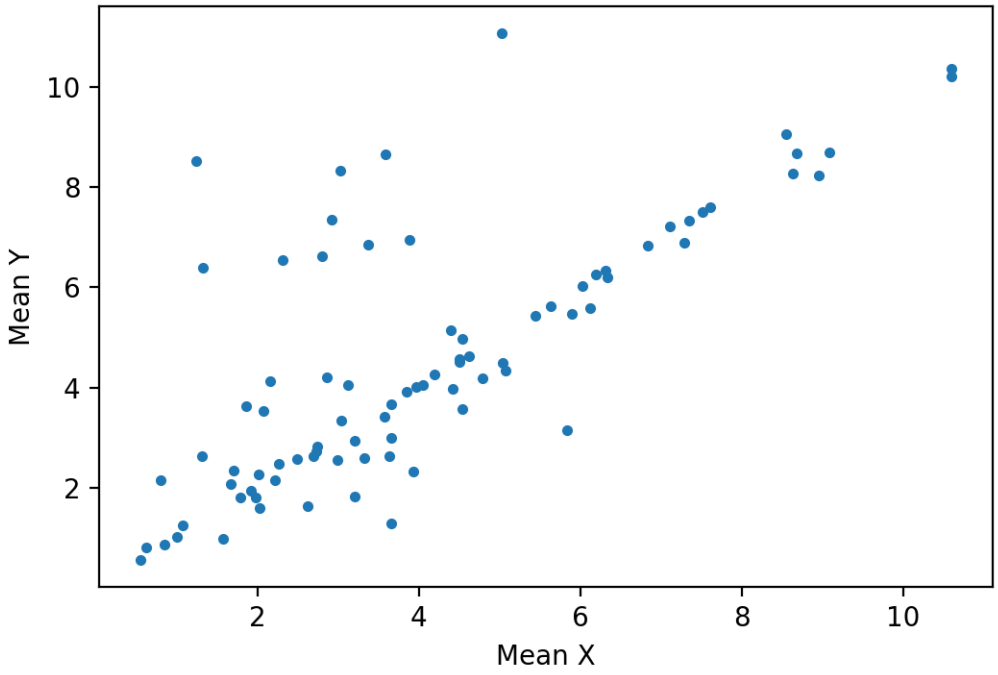


Figure 8.2: Scatter plot of the Mean X and Y values.

If we want to just get the values from a column into a data structure we have worked with before, we can do the following:

```
In [6]: df.Mean_X.values.tolist()
```

```
Out[6]: [5.031000000000001,
         10.599,
         10.589,
         8.548,
         9.089,
         8.675,
         3.588,
         1.244,
         3.023,
         8.632,
         ...
```

Where `values` gets all the values in the column Mean_X, and `tolist()` converts the column into a list.

If we want to grab a specific row within the data frame, we can use the function `loc`, which uses the index (in our case just row numbers starting at 0).

```
In [7]: print(df.loc[0])
        print()  # create a blank line in output
        print(df.loc[50])
        print()  # create a blank line in output
        print(df.loc[50,"Mean_X"])
```

```
Name                    XMMXCS J113313.8+662243.9
Redshift                                     0.12
Mean_X                                      5.031
Mean_x_minus_delta                          4.805
Mean_x_plus_delta                           5.255
Mean_y                                     11.087
Mean_y_minus_delta                         10.239
Mean_y_plus_delta                          11.883
Name: 0, dtype: object

Name                    XMMXCS J004252.6+004303.1
Redshift                                     0.27
Mean_X                                      2.072
Mean_x_minus_delta                          1.496
Mean_x_plus_delta                           3.187
Mean_y                                      3.527
Mean_y_minus_delta                          3.002
Mean_y_plus_delta                           4.525
Name: 50, dtype: object

2.072
```

Here we see all the information for the first row (row 0) in the first print statement; use a lazy way of forcing a new line by calling the print function without any arguments; see all the information for the 51st row and then only print the value from the "Mean_X" column of the 51st row by passing the column name to the `loc` function as the second argument.

The `loc` function also allows slicing, similar to the way you have previously done for lists.

```
In [8]:  print(df.loc[0:2,"Mean_X"])
         print(df.loc[0:2,"Mean_X"].values.tolist())
```

```
0     5.031
1    10.599
2    10.589
Name: Mean_X, dtype: float64

[5.031000000000001, 10.599, 10.589]
```

Where we have used the `values.tolist()` functions to turn 3 rows from the column into a list, which we are more familiar with.

It is encouraged that you use the **help** function to look through `pd.DataFrame`, which will tell you about all the other functions which can be used. In particular the `shape` function can be used to tell you how many rows, and how many columns are in the data frame.

```
In [9]:  df.shape
```

```
Out[9]: (84,8)
```

The output, **(84,8)**, tells us that our `DataFrame` (`df`) has 84 rows and 8 columns. Note that this means we actually have 84 rows, not 83 rows and the columns names; i.e. the column names **do not** count as a row.

You can apply all operators to the columns, and pandas will perform the operation on each row at a time. You can then store the results back into the data frame in the same way as a dictionary.

```
In [10]: # Define new columns containing the data produced by an operation
         df["Mean_X_upper"] = df.Mean_x_plus_delta - df.Mean_X
         df["Mean_X_lower"] = df.Mean_X - df.Mean_x_minus_delta

         # Output the first row after making these changes
         df.loc[0]
```

```
Out[10]: Name                 XMMXCS J113313.8+662243.9
         Redshift                                 0.12
         Mean_X                                  5.031
         Mean_x_minus_delta                      4.805
         Mean_x_plus_delta                       5.255
         Mean_y                                 11.087
         Mean_y_minus_delta                     10.239
         Mean_y_plus_delta                      11.883
         Mean_X_upper                            0.224
         Mean_X_lower                            0.226
         Name: 0, dtype: object
```

Saving your DataFrame

Before we move onto the exercises and analysing the data from the csv, you may want to save your dataframe in some manner. To do so you can write the DataFrame back to a csv - creating a hard copy so to speak. This way - if you screw up - you can load back in the data from a previous state. For this course, you will rarely be working on sufficiently large data to warrant such an approach, but it is useful.

To save as a csv file simply call the `to_csv` method.

```
In [11]  df.to_csv("Week6_with_errors.csv")
```

8.2.4 Exercises

Have you managed to get the first 10 entries of the astronomy file to print out? If not, don't move on until you've done it.

1. Week6Basic - Write a code to view and create columns in a DataFrame.

   a) Print the 50th, 72nd and 9th rows, including all their columns.

   b) In one code line, print the Mean_y values for **rows** 62-71.

   c) Add two new columns to your data frame similar to the Mean_X_upper and Mean_X_lower example above for y. (You should also have done this for X, if not, do this now).

   d) Save the updated DataFrame with the new columns to a csv file.

   e) Read the csv you just created back into a new DataFrame, and have a look - what do you now notice? (If you're not sure look at pd.read_csv with the help function, and if you're still not sure ask an AT)

8.3 Plotting data points and error bars

8.3.1 Data points

Last week you were introduced to some basic plotting with the matplotlib library. Already this week we have used an example of the scatter plot, and for the rest of this week we will be going through some basic tools for handling and fitting data in python.

To begin with we are going to create some random data, and create a scatter plot. Unlike last week's section on this, here we will use the `numpy.random` module. This is covered again in Sec. 9.5.

```
In [12]  from numpy.random import default_rng

         # Set the seed so that we all get the same result.
         rng = default_rng(12345)

         # Create random data for x and y
         x_data = rng.random(100)
         y_data = rng.random(100)

         # Plot the random data
         plt.figure()
         plt.scatter(x_data,y_data,marker='x',color='r')
         plt.show()
```

Hopefully you should get something that looks like Figure 8.3.

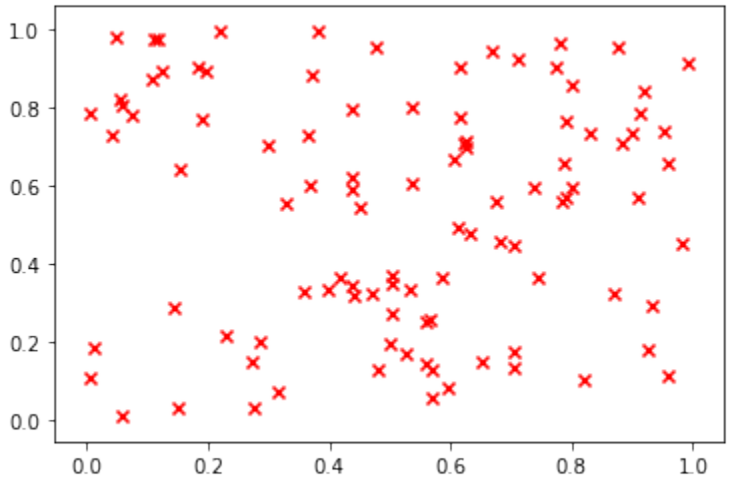


Figure 8.3: Scatter Plot result of random points.

You are welcome to (and for some of the exercises required to) use the table below for reference. It contains a selection of possible marker and color options available to most `matplotlib` functions.

b	blue
g	green
r	red
c	cyan
m	magenta
y	yellow
k	black
w	white

s	square
p	pentagon
*	star
h	hexagon
o	circle
+	plus
x	cross
D	diamond
-	line

Table 8.2: A reference for plotting data points with markers and colours.

8.3.2 Plotting error bars

`Matplotlib` allows you to plot error bars using the `errorbar` function in place of `plot` or `scatter`. Here we will show you a few examples using `errorbar`.

- **Example 1:** the error bars are symmetrical and the same on each point, see Figure 8.4.

```
In [13]  plt.figure()
         plt.errorbar(x_data,
                      y_data,
                      yerr=0.05,
                      xerr=0.05,
                      fmt='ro',
                      ecolor="k",
                      capsize=2)
         plt.show()
```

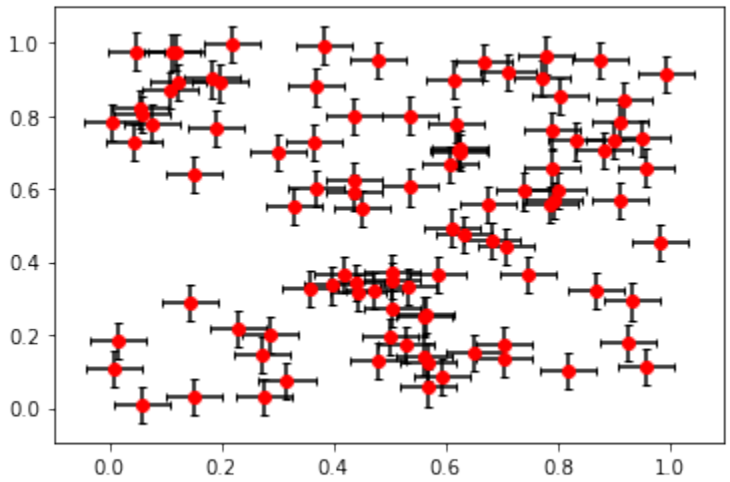


Figure 8.4: Scatter Plot result of random points with error bars.

For this example, we have used the `yerr` and `xerr` arguments to plot symmetric error-bars on each point. `fmt` is the marker format argument, where `r` denotes the color red, and the `o` tells `errorbar` to use circles as markers (these can be exchanged for the `color` and `marker` respectively). For the errorbar color, we use the argument `ecolor`, which uses the same color names as the normal plotting functions you've already come across. `capsize` is an optional choice, that sets the size of the little orthogonal bars on the end of each error bar.

- **Example 2:** the error bars are symmetrical, but in the x-direction they are different for each point, see Figure 8.5.

```
In [14]  # Get random errors for the x axis
         xerr = rng.random(100)/10

         # Plot the result
         plt.figure()
         plt.errorbar(
                     x_data,
                     y_data,
                     yerr=0.05,
                     xerr=xerr,
                     fmt='ro',
                     ecolor="k",
                     capsize=2)
         plt.show()
```

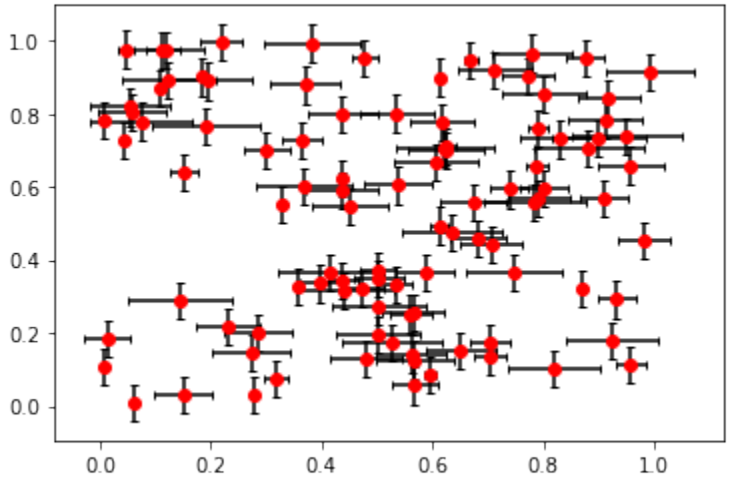


Figure 8.5: Scatter Plot result of random points with constant y and random symmetric x error bars.

- **Example 3:** the error bars are not symmetrical (they have different upper and lower values) and differ for each point. See Figure 8.6.

```
In [15]  # Get random upper and lower errors for both x and y
         xerr = [rng.random(100)/10,
                 rng.random(100)/10]
         yerr = [rng.random(100)/10,
                 rng.random(100)/10]

         # Plot the result
         plt.figure()
         plt.errorbar(
                     x_data,
                     y_data,
                     yerr=yerr,
                     xerr=xerr,
                     fmt='ro',
                     ecolor="k",
                     capsize=2)
         plt.show()
```

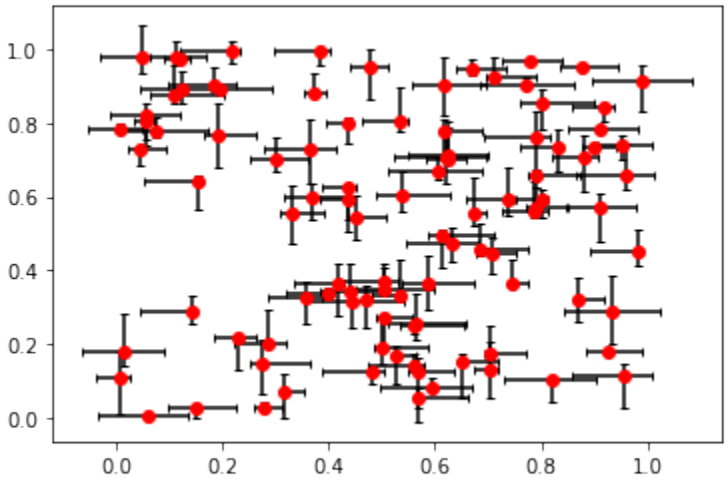


Figure 8.6: Scatter Plot result of random points with random error bars.

Exercises (with worked answers)

See Section 8.5 for worked answers.

1. Create a range from 0 to 5 with steps of 0.01 and assign to variable t. Then apply the function $f(t) = t\sin(10t)$ and plot with data points as yellow stars.

2. Plot the following data with its associated errors, with the data point as blue diamonds and error bar colour as green:
   y=1.474, 1.021, 0.506
   x=0.00400, 0.00270, 0.00120
   x_lower=0.001, 0.0002, 0.0003

x_upper=0.001, 0.001, 0.001
y_lower=0.003, 0.002, 0.003
y_upper=0.004, 0.004, 0.01

3. Using your DataFrame from Section 8.2.4, plot X against Y with your calculated error bars. Limit the X axis between 0 and 12

Exercises (other)

1. Plot $y = \sin(x)$ with the x being a range from 0 to 5 with increments of 0.1, with y-error = 0.5, x-error = 0.

2. Plot $y = \sin(x)$ with the x being a range from 0 to 5 with increments of 0.1, with y-upper-error = +0.5, y-lower-error = −2, x-error = 0. With the data points being in red and the error bar colour green.

3. Using a file containing Exoplanet data `PythonExampleCSVfile2.csv`, plot Semi-Major axis (AU) vs. Orbital Period (day). Apply associated errors, have the error bars as black and data points as red pluses. Label axis and add a title 'Exoplanet data'. Table 8.3 shows the layout of the csv file.

Table 8.3: Contents of exoplanets data csv

Column A:	name
Column B:	X (Semi-Major axis)
Column C:	X error
Column D:	Y (Orbital period)
Column E:	Y lower error bound
Column F:	Y upper error bound

8.4 Fitting straight lines to data points

There are numerous ways of fitting lines to data in Python. A more sophisticated method will be covered in Chapter 10 but we will show here a simple method that utilises the numpy.polynomial module. This module allows one to fit polynomials of varying order. The example below shows how to fit a 1D polynomial.

```
In[16]  from numpy.polynomial import Polynomial
        # Mock data to fit
        x = [1, 2, 3, 4, 5, 6, 7, 8]
        y = [1.5,2,3.6,4,5,6.7,7.45,8.1]

        # Using numpy.polynomial, fit the line with a first order polynomial
        p = Polynomial.fit(x, y, deg=1)

        # Some x values to plot the fit with
        x_fit = np.linspace(0, 10, 100)

        # And plot the results
        plt.figure()
        plt.plot(x, y, 'ro', label='Data' )
        plt.plot(x_fit, p(x_fit), label='Best fit line')
        plt.legend(loc='best')
        plt.show()
```

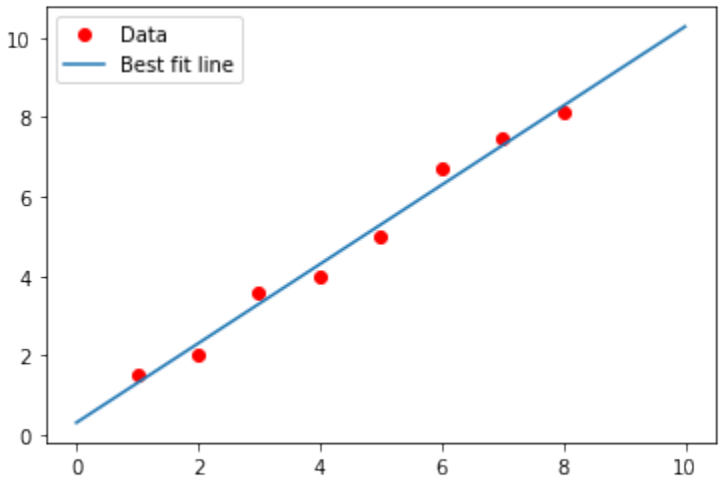


Figure 8.7: Plot result from fitting the mock data in Section 8.4

See Fig.8.7 for the output. Note that here we are using the label argument to add labels to the fit and data points, and the legend method to add a legend to the plot with these labels. Lets go through the rest of the example:

- We define some x and y data as lists.

- Next we call the `Polynomial.fit` function which uses the raw `x` and `y` data to compute an instance `p` of a Polynomial of degree 1. If you wish to see the output of `poly1d` use `print(p)`, and the following output should be shown `4.79 + 3.49 x`.
- The `linspace` function creates an array to evaluate the fit at (variable `p` contains the fitting function). The general syntax for `linspace` is `linspace(a,b,n)`, where the array runs from `a` to `b` (**inclusive of `b`**) using an `n` number of points.
- The `legend` command in the next line uses the labels from the plots and creates a legend. We use the `loc` argument to define the legend's location, in this case `'best'` means it places the legend in the best location. The default location is indeed `'best'`, but as previously mentioned it's good to be specific!
- The final command shows the plot.

8.4.1 Exercises

1. Plot x vs. y with red circles as data points. Use `polyfit` to fit a line of best fit as a green dashed line and add a legend. Hint: For dashed line use `'--'`. .
   x = 0.0, 0.5, 1.0, 1.5, 2.0, 2.5, 3.0, 3.5, 4.0, 4.5, 5.0, 5.5, 6.0, 6.5, 7.0, 7.5, 8.0, 8.5, 9.0, 9.5.
   y = -0.23, 0.53, 3.8, -3.16, -7.45, -11.76, 2.38, -14.65, 12.64, -13.2, 2.99, 21.08, -3.48, 26.74, -19.49, 8.03, -4.63, -49.06, 29.73, 54.69
2. Fitting noisy data:
   a) Plot $f(x) = 2x^2 + 6$ in the range $x \in [-5, 5]$.
   b) Add random Gaussian noise to $f(x)$ with $\mu = 0, \sigma = |x|$ without using loops.
   c) Fit a 2nd order polynomial to your altered $f(x)$.
   4) Plot the residuals between your fit and $f(x)$.

8.5 Answers to Exercises from Sec 8.3.2

This section assumes a blank notebook for each answer. You will not need to import the modules every time.

```
In[1]  import matplotlib.pyplot as plt
       import numpy as np

       # store values as asked
       t = np.arange(0,5,0.01)

       # Create the desired expanding sine wave
       f = lambda t: t*np.sin(10*t)

       # Finally plot the function as yellow stars
       plt.figure()
       plt.scatter(t, f(t),marker='*',color='yellow')
       plt.show()
```

```
In[1]  import matplotlib.pyplot as plt

       # Store the data from the question as lists
       y=[1.474, 1.021, 0.506]
       x=[0.00400, 0.00270, 0.00120]
       xlower=[0.001, 0.0002, 0.0003]
       xupper=[0.001, 0.001, 0.001]
       ylower=[0.003, 0.002, 0.003]
       yupper=[0.004, 0.004, 0.01]

       # Plot with errors as blue diamonds, and green error bars.
       plt.figure()
       plt.errorbar(
               x,
               y,
               xerr=[xlower,xupper],
               yerr=[ylower,yupper],
               fmt='bD',
               ecolor='g',
               capsize=2)
       plt.show()
```

```
In [1]  import pandas as pd
        import matplotlib.pyplot as plt

        # Load in the data
        df = pd.read_csv("PythonExampleCSVfile1.csv",index_col=None)

        # Calculate the upper and lower errors
        df["Mean_X_upper"] = df.Mean_x_plus_delta - df.Mean_X
        df["Mean_X_lower"] = df.Mean_X - df.Mean_x_minus_delta
        df["Mean_y_upper"] = df.Mean_y_plus_delta - df.Mean_y
        df["Mean_y_lower"] = df.Mean_y - df.Mean_y_minus_delta

        # Finally plot the results
        # Note here that .T simply transposes the result, but any
        # other method is just fine (even calling each column
        # individually is valid)
        plt.figure()
        plt.errorbar(
                df.Mean_X,
                df.Mean_y,
                xerr=df[["Mean_X_lower","Mean_X_upper"]].values.T,
                yerr=df[["Mean_y_lower","Mean_y_upper"]].values.T,
                fmt='bo',
                ecolor="k",
                capsize=3)
        plt.xlim(0,12)
        plt.show()
```

Chapter 9
Numpy and Arrays

9.1 Learning Outcomes

By the end of today's session you will know how to:

1. Create and manipulate `numpy` arrays.
2. Perform basic calculations and operations using arrays.
3. Generate arrays containing random values.
4. Perform matrix operations with `numpy`.
5. Use the `where` function.
6. Utilise `numpy` to optimise programs (advanced).

9.2 What is `numpy`?

You have seen some of `numpy` in the previous chapters but this has barely scratched the surface of what `numpy` really is. It is in fact one of the most powerful and frequently used modules available in `Python`. It contains basic mathematical functions, a whole host of common linear algebra operations, a data structure that makes many complex operations simple, and random number capabilities to name a few. Some of its abilities replicate functionality that already exists in the base `Python` language, but normally the `numpy` implementation will be much faster.

This chapter will not be comprehensive, `numpy` is such a large module that teaching you how to use it all would take a course by itself, but it will cover the most useful and important features. Full documentation can be found at `https://docs.scipy.org/doc/numpy/`.

The main building block of `numpy` is the array (ndarray to be specific), which acts somewhat like multi-dimensional lists. In fact you may have seen similar structures before: a list of lists is inherently a 2-D object with rows (the index on the inner list) and columns (the index on the outer list), however lists of lists like these have serious limitations - such as not

being able to extract individual columns. Arrays however not only allow you to extract rows and columns but contain a large number of useful features. Below is a demonstration of some basic operations.

One way to define an array is to take an existing list and convert it using `np.array()`.

```
In [1]: # Define a list
        list1 = [0, 1, 2, 3, 4]

        # Convert this list to a 1-D numpy array
        arr1d = np.array(list1)
```

We can then perform a number of basic statistical operations.

```
In [2]: # Compute the mean
        np.mean(arr1d)

        # Compute the median
        np.median(arr1d)

        # Compute the sum
        np.sum(arr1d)

        # Compute the standard deviation
        np.std(arr1d)

        # Compute the variance (square of standard deviation)
        np.var(arr1d)
```

The argument of these functions does not have to be an array however, in all the statements above `arr1d` could be exchanged with `list1` and you would get the same outputs. However, by using arrays, most of these can be utilised with a different syntax:

```
In [3]: # Compute the mean
        arr1d.mean()

        # Compute the median
        np.median(arr1d)

        # Compute the sum
        arr1d.sum()

        # Compute the standard deviation
        arr1d.std()

        # Compute the variance (square of standard deviation)
        arr1d.var()
```

Both of the above blocks will produce the following output if you print each operation.

```
Out[3]: 2.0
        2.0
        10
        1.4142135623730951
        2.0
```

Notice most of these results have been converted from the input integers to floats with the exception of the sum which has maintained the integer dtype. This will happen unless a data type argument (dtype) is stated within the `np.array()` call or sum, mean or std etc.

We can also extract meta data from the array such as it's size (similar to `len`).

```
In [4]: # Get the size of the array
        arr1d.size
```

```
Out[4]: 5
```

Notice the lack of parentheses on the size call. This is important since size is not 'callable', i.e. it is not a function but instead a property of the array. This will make more sense if you continue on to using object orientated `python` after this module.

9.3 Array creation

There are a number of ways to create an array in `numpy`, the exact use normally dictating which is the preferred method. We have already covered converting a list to an array but there are 3 useful ways to create an array, particularly if you need an array of a certain size to populate in a loop or if you need certain elements to be initialised.

The first of these is `np.empty`, which initialises an 'empty' array. 'Empty' here meaning an array where the contents are not initialised at any predictable value.

```
In [1]: # Create an empty array
        empty_arr = np.empty((4, 4))
```

This creates a 4x4 array of 'empty' values. Another and more often used version produces an array of zeros.

```
In [2]: # Create an array of zeros
        zero_arr = np.zeros((4, 4))
```

Again, this creates a 4x4 array but this time filled with 0. Finally, you can create an array filled with a particular value.

```
In [3]: # Create an array of the value of pi
        full_arr = np.full((4, 4), np.pi)
```

This will create a 4x4 array filled with the value of pi. In each of these cases the argument you pass the function is the shape of the array, if more than 1D this must be a tuple as shown here. In the `full` case you must also supply the second argument which is the value you wish the array to be filled with. Notice also this array has be filled with np.pi, this is the value of π which `numpy` has stored as an attribute. Don't worry too much about the semantics of this statement, just remember pi can be accessed from numpy with `np.pi` without brackets (again an attribute/property, not a callable).

In addition to the methods shown above each of these 3 array creation methods has a `_like` version which will create an array using the shape of another like so.

```
In [4]: # Define a list
        list1 = [[0, 1], [2, 3], [4, 5]]

        # Convert this list to a 2-D numpy array
        arr2d = np.array(list1)
        print(arr2d.shape)

        # Create an empty array
        empty_arr = np.empty_like(arr2d)

        # Create an array of zeros
        zero_arr = np.zeros_like(arr2d)

        # Create an array of the value of pi
        full_arr = np.full_like(arr2d, np.pi)
        print(empty_arr.shape, zero_arr.shape, full_arr.shape)
```

```
Out[4]: (3, 2)
        (3, 2) (3, 2) (3, 2)
```

These methods are summarised in Tab.9.1.

Function	Arguments	Description
`np.array`	list, dtype (optional)	Converts a list (or list of lists) to an array with optional ability to specify data type for elements.
`np.empty`	shape	Creates an empty array filled with uninitialised values.
`np.empty_like`	array	Same as `np.empty` but uses the shape of another array.
`np.zeros`	shape	Creates an array of zeros of the given size.
`np.zeros_like`	array	Same as `np.zeros` but uses the shape of another array.
`np.full`	shape, fill_value	Creates an array filled with a specified value.
`np.full_like`	array, fill_value	Same as `np.full` but uses the shape of another array.

Table 9.1: A table showing a selection of ways an array can be created using `numpy`. Note this is note exhaustive, nor are the arguments a complete list of the possible arguments, for a complete list of arguments see `https://docs.scipy.org/doc/numpy/`.

9.4 Computations with Arrays

Once we have created arrays we can do a number of computations and manipulations with them. You have seen some of these in Chapter 7. Here we cover mathematical operators to display array behaviour but `numpy` also contains a huge bank of mathematical functions (such as sin, log, exp etc., note that in `numpy` `log` is the natural log and `log10` is log in base 10). To find your desired function either use `dir` or look in the documentation.

Any mathematical operators can be used on the array as a whole (operating on each individual element in a hidden loop in C), which is much faster than if this were done using a `python` loop. We can show this with a simple example

```
In [1]: # Define a list
        list1 = [[0, 1], [2, 3], [4, 5]]

        # Convert this list to a 2-D numpy array
        arr2d = np.array(list1, dtype=float)

        # Multiply the elements by 4
        arr2d *= 4
        print(arr2d)

        # Add 4
        arr2d += 4
        print(arr2d)

        # Subtract 12
        arr2d -= 12
        print(arr2d)

        # Divide by 2
        arr2d /= 2
        print(arr2d)
```

```
Out[1]: [[ 0.  4.]
         [ 8. 12.]
         [16. 20.]]
        [[ 4.  8.]
         [12. 16.]
         [20. 24.]]
        [[-8. -4.]
         [ 0.  4.]
         [ 8. 12.]]
        [[-4. -2.]
         [ 0.  2.]
         [ 4.  6.]]
```

We can also multiply arrays as long as they 'have the same dimension', i.e. in the simple case they are the same shape. This multiplies together the entries sharing the same position in each array.

```
In [2]: # Define a list
        list2 = [[5, 4], [3, 2], [1, 0]]

        # Convert this list to a 2-D numpy array
        arr2d2 = np.array(list2)

        # Multipply the array by another array
        arr2d *= arr2d2
        print(arr2d)
```

```
        [[-20.  -8.]
         [  0.   4.]
         [  4.   0.]]
```

You can test the other operations for your self but all of the above operations will work.

The important thing to note is that this works if the array is the same shape OR must has the same shape as the final dimension. In other words, an array containing N 2-element vectors can be multiplied by either a $N \times 2$ array or a 2-element vector. This is shown below.

```
In [3]: # Create a 2 element vector
        vec2 = np.array([5, 5])

        # Multiply the array by a 2 vector
        arr2d *= vec2
        print(arr2d)
```

```
        [[-100.  -40.]
         [   0.   20.]
         [  20.    0.]]
```

Again you can check this works with all other operations.

We can also find the maximum and minimum of an array as well finding the index of the maximum or minimum.

```
In [4]: # Define an array
        arr = np.array([1, 2, 3, 4, 5])

        # Get the maximum and it's location
        arr.max()  # value
        arr.argmax()  # index

        # Get the minimum and it's location
        arr.min()  # value
        arr.argmin()  # index
```

```
        5
        4
        1
        0
```

9.4.1 Exercises (with worked solutions)

1. Create an array of zeros. Loop over it storing the square of the index as each element.
2. Create an array full of -2. Multiply all elements by -2.
3. Find the sum, mean, standard deviation, and median of the array from the first exercise.

9.4.2 Broadcasting

How arrays behave when operated on with another array is dictated by `numpy`'s "broadcasting rules" (for instance multiplying two together). Rather than give a long explanation of this here we instead want to draw your attention to the `numpy` documentation. This is a fantastic resource and has a very good page explaining the "broadcasting rules", this can be found at: `https://docs.scipy.org/doc/numpy/user/basics.broadcasting.html`.

Keep in mind that this documentation exists, there is a full description, and often examples, of all functions within `numpy` and also `scipy` which you will see more of next week. We have mentioned this a fair amount already but the internet is a valuable resource with both documentation and forum posts available for most widely used packages.

9.4.3 Extraction and Shape/Dimension Manipulations

Just like lists, elements can be extracted from arrays by indexing with the position of the desired element.

```
In [1]: # Define a 1D array
        arr = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])

        print(arr[4])
```

```
        5
```

Of course unlike lists an array can have multiple dimensions in which case indexing for a single element needs have an index for each dimension.

```
In [2]:# Define a 2D array
        arr2D = np.array([[1, 2, 3, 4, 5], [6, 7, 8, 9, 10]])

        print(arr2D[1, 4])
```

```
        10
```

Again like lists, slices can be extracted from an array.

```
In [3]: print(arr2D[1, 1:4])
```

```
Out[14]: [7 8 9]
```

Having multidimensional arrays means you can index in all sorts of ways. Slicing along certain axes/dimensions or extracting particular rows, columns or any combination of these. The following example shows extracting the first 10 rows of an array and then extracting the last 10 rows but only the first column of those rows.

```
In [4]: # Define a 2D array
        arr2d = np.array([[1, 2], [3, 4], [5, 6], [7, 8], [9, 10], [11, 12],
                          [13, 14], [15, 16], [17, 18], [19, 20], [21, 22],
                          [23, 24]])

        print(arr2d[:10])
        print(arr2d[-10:, 0])
```

```
        [[ 1  2]
         [ 3  4]
         [ 5  6]
         [ 7  8]
         [ 9 10]
         [11 12]
         [13 14]
         [15 16]
         [17 18]
         [19 20]]
        [ 5  7  9 11 13 15 17 19 21 23]
```

You can generalise this to a greater number of dimensions than 2 as you please.

One huge advantage of arrays over lists is that they can accept a list or an array of indices to extract specific elements! The `numpy` module is so proud of these indexing properties they call the process "fancy indexing". This not only refers to the ability to provide lists of indices, which is a necessity with multidimensional arrays, but also to the fact that the result of a fancy index (array indexing) is the same shape as the index not the array being indexed, a not immediately obvious subtlety.

```
In [5]: print(arr2d[[0, 4, 9, 10]])
        print(arr2d[(0, 1, 4, 6), (0, 1, 0, 1)])
```

```
        [[ 1  2]
         [ 9 10]
         [19 20]
         [21 22]]
        [ 1  4  9 14]
```

Of course this isn't the end of the story... arrays are wonderful things with a seemingly endless number of useful features and surprising behaviours. One of these is "Boolean indexing". This allows you to index based on a conditional; elements for which the condition is True are returned and elements for which the condition is False are not.

```
In [6]: small_1d = arr2d[0:5,1]
        print(small_1d)
        print(small_1d[[True, False, True, True, False]])
        print(small_1d[small_1d > 5])
```

```
         [ 2  4  6  8 10]
         [2 6 8]
         [ 6  8 10]
```

As well as extracting elements we can also manipulate arrays in a number of ways. There are `numpy` methods which will flip arrays, these can be useful when array shapes get complex but there's a really simple way to reverse the order along a axis of an array using indexing:

```
In [7]: print(small_1d[6:1:-1])
        print(small_1d[::-1])
```

```
         [10  8  6]
         [10  8  6  4  2]
```

To fully reverse the order of a multidimensional array you can simply do this along each dimension.

You can also reshape an array. Now lets say you have an array of shape (10, 10) but you want is to be of shape (5, 20) this can be done by using .reshape(shape), where shape is a tuple containing the desired shape.

```
In [8]: arr3 = np.full((10,10),2)
        print(arr3.shape)
        arr3 = arr3.reshape(5,20)
        print(arr3.shape)
```

```
        (10, 10)
        (5, 20)
```

It's important to note how this is performed by `numpy`. What reshape does effectively is run through the elements of an array filling the new shaped array with elements 1 by 1 going along rows. If using reshape be sure elements are ending up in the desired position.

9.5 Using `numpy`'s Random Functions

In Chapter 7 we showed you the `random` module. There is absolutely nothing wrong with the `random` module but for certain applications the implementation of very similar functions in `numpy.random` (briefly introduced in Section 8.3.1) is much better suited to the job (one upside of the `random` module is that it is "thread safe", but this will only matter to you if you find your self doing any parallel computing later on).

The main difference between `random` and `numpy.random` is that `numpy.random` has the ability to produce multiple random numbers at once in an array.

In much the same vein as when we presented the `random` module, do the following exercises using both `dir` and `help`, and the `numpy` docs: `https://numpy.org/doc/stable/reference/random/index.html`. Learning how to work from documentation is essential not only if you become a programmer later on in your career but also to complete many of the assignments you will be set in later years.

9.5.1 Exercises

Exercises (with worked solutions)

1. Generate 100 random numbers between 0 and 1.
2. Generate a 2D array of shape (100, 100) with random integers between 1 and 50.
3. Generate 1000 random numbers pulled from a beta distribution with $a = 1$ and $b = 100$.
4. Generate 10000 random numbers from a normal distribution with $\mu = 5$ and $\sigma = 2$.

Advanced Exercises

1. Generate 100 random integers between 1, 10. Can you count how many instances of each number arise? (hint: could a dictionary make this trivial? Alternatively does `numpy` have something to do this in 1 line?)

2. Do the same as for the previous questions but pulling from a normal distribution with mean $\mu = 100$ and $\sigma = 20$ and also for a Poisson distribution with the same mean. You will have to manually convert the floats returned by these random distributions to integers unless you want to properly bin the data.

3. Plot the resulting binned counts (a histogram) from the normal and Poisson distribution. What happens if you increase the number of samples?

9.6 Treating Arrays as Matrices

`Numpy` also contains a sub-module (like `numpy.random` is a sub-module within `numpy`) for linear algebra, namely `numpy.linalg`. This, as you may have guessed, is a powerful sub-module for performing matrix operations on arrays. There is a lot within `numpy.linalg` but here we will give a quick run down of some of the more commonly used operations.

Lets define 2 arrays.

```
In [1]: arr1 = np.array([[1,2],[3,4]])
        arr2 = np.array([[4,3],[2,1]])
```

We can then perform the dot product of these arrays.

```
In [2]: np.dot(arr1, arr2)
```

```
Out[2]: array([[ 8,  5],
               [20, 13]])
```

We can also extract the eigenvalues of one these matrices (arrays).

```
In [3]: np.linalg.eigvals(arr1)
```

```
Out[3]: array([-0.37228132,  5.37228132])
```

Or compute the trace.

```
In [4]: np.trace(arr1)
```

```
Out[4]: 5
```

Admittedly this is a very quick set of examples but beyond these `numpy.linalg` contains many many functions including advanced linear algebra functions and even tensor operations used in the toolkit of General Relativity.

One of the more often used functions is the `norm` function. This computes the norm of an array but is particularly useful when calculating the radii of a bunch of 2 or 3 dimension coordinates. Lets treat the matrix above as an array of coordinate vectors.

```
In [5]: # Get the norm of the whole array
        print(np.linalg.norm(arr1))

        # Compute the radius of each coordinate
        print(np.linalg.norm(arr1, axis=1))
```

```
        5.477225575051661
        [2.23606798 5.        ]
```

Again this is just a quick run down of some common functions but should you find yourself doing any heavy lifting with linear algebra in theory or anything involving positions in simulations, you may find yourself using these functions often.

9.7 The `Where` Function

Lets imagine you have a very large array of values but you actually only care about a subset above a certain value threshold. You can easily do this by using Boolean indexing.

```
In [6]: arr = rng.normal(10,2,1000)
        arr[arr > 15]
```

```
Out[6]: array([15.52768816, 15.27558242, 15.0953834 ,
               15.30399824, 15.28584537])
```

However, sometimes you may want to actually extract the indices themselves, this is where the `where` function comes in. The `where` function does exactly what it says on the tin, "where is the condition true".

```
In [7]: np.where(arr > 15)
```

```
Out[7]: (array([237, 303, 700, 704, 825]),)
```

This is very very useful but does have its limitations when arrays begin to get **very** large (more than 10000000 elements on a modern Macbook Pro), here `where` becomes slower and there are other coding paradigms that begin to become a necessity at this scale.

Although this is often how `where` is used this is actually a slight misuse of the function. The power of where lies in applying an operation or function to an element in an array where the condition is True. Doing this the syntax of `where` is `np.where(condition, array_to_apply_to, operation)`.

```
In [3]: # Define an arrays from lists
        x = np.array([[1, 2], [1, 3], [4, 5], [6, 5], [9, 6]])
        y = np.array([[3, 2], [5, 4], [1, 1], [7, 6], [8, 3]])

        # Add 10 to any value below 4
        np.where(x < 4, x + 10, x)

        # The operation can be performed on a different array based on
        # the condition applied to the original array or some compound
        np.where(x - y < 4, y, y + 10)
```

Do note that `where` needs to be assigned to a new variable to maintain the changes, it is not performed inplace on the original array, instead creating a copy.

9.7.1 Exercises (with worked solutions)

1. Create a random array of numbers between 1 and 0. Use `where` or Boolean indexing to find how many lie above 0.5 and how many lie below.

2. Take the array from the previous question and add 0.5 to any numbers below 0.5, check this has worked by ensuring there are no numbers below 0.5.

3. Create 2 new arrays of numbers between 1 and 0. Add 0.5 to elements of array 1 where array 2 is less than 0.5. How many numbers lie below 0.5 in array 1?

9.7.2 Advanced Exercises

1. Create 2 arrays of 1000 random numbers from a normal distribution with $\mu = 5$ and $\sigma = 2$. Plot them as a scatter plot with '+' markers. Then plot over the top of this plot with 'o' markers smaller than the '+' markers colouring each quadrant a different colour (hint: use `where`).

9.8 Optimisation with numpy

It's already been mentioned in passing but one of the single best things about `numpy` is... IT'S FAST! A clever implementation of `numpy` can take code that takes hours down to minutes or even seconds. This is no exaggeration either, I have witnessed this first hand!
The reason for this is two-fold, firstly `numpy` is implemented in C, this makes it inherently faster at performing basic operations like multiplication of arrays but secondly it is "vectorised". Essential this means removing loops. What `numpy` is actually doing when you vectorise, is move the slow `python` loop into C which is executed much faster.
We can probe this with an example utilising `%timeit`, this allows you to define functions and time them to compare implementations and optimise your code. How fast your code can run can really begin to matter, either you're writing release code which needs to run in a certain time or research code which may work with "Big Data" meaning you need to write code which needs to be fast to actually finish in a reasonable time.
An example of this is shown below multiplying a single array by 5.

```
In [1]: # Define functions to test (simple with a lambda function)
        vec_func = lambda x: x * 5

        def list_func(x):
            # Loop over elements multiplying by 5
            for ii, i in enumerate(x):
                x[ii] = i * 5
            return x

        # Define an array and list to test with
        xs = rng.normal(10, 3, 1000)
        ys = list(xs)

        # Time the implementations
        %timeit vec_func(xs)
        %timeit list_func(ys)
```

```
    1.57 us +/- 78.9 ns per loop (mean +/- std. dev. of 7 runs,
↪ 1000000 loops each)
    302 us +/- 30.7 us per loop (mean +/- std. dev. of 7 runs, 1000
↪ loops each)
```

Notice that not only is the `numpy` implementation 100 times faster on this simple operation, it is also far more consistent with a standard deviation of the order of nanoseconds! This is a very contrived example but well displays how effective an intelligent `numpy` implementation can be.

This marks the end of this introductory `numpy` chapter. Next chapter we will look into applications of what you have learnt to real data which will contain some more (and more interesting parts) of `numpy`.

9.9 Worked Solutions

9.4.1

```
In [1]  # Question 1
        arr = np.zeros(10)
        for index in range(10):
            arr[index] = index**2

        # Question 2
        arr2 = np.full(10,-2) * -2

        # Question 3
        arr.sum()
        arr.mean()
        arr.std()
        np.median(arr)
```

9.5.1

```
In [2] # Import rng
       from numpy.random import default_rng
       rng = default_rng()

       # Question 1
       rng.random(100)

       # Question 2
       rng.integers(1, 50, (100, 100))

       # Question 3
       rng.beta(1, 100, 1000)

       # Question 4
       rng.normal(5, 2, 10000)
```

9.7.1

```
In [3]  # Question 1
        arr = rng.random(100)
        print(np.where(arr > 0.5)[0].size)
        print(np.where(arr < 0.5)[0].size)

        # Question 2
        arr2 = np.where(arr < 0.5, arr + 0.5, arr)
        print((arr2 < 0.5).sum())
```

```
# Question 3
arr1 = rng.random(100)
arr2 = rng.random(100)

arr3 = np.where(arr2 < 0.5,arr1 + 0.5, arr1)
(arr3 < 0.5).sum()
```

Chapter 10
Scientific Python

10.1 Learning Outcomes

By the end of today's session you will know how to:

1. Open a range of different file formats in `Python`.
2. Use arrays as images, manipulating and plotting them.
3. Compute histograms and plot them.
4. Perform simple fitting on data.
5. Perform simple integration using `scipy`.

In this chapter we will try to demonstrate some of the uses of `Python` you will come across in the future, either in modules or research projects. Particularly, in Year 2 you will have a module called Scientific Computing, this module is designed to teach you the theory behind many scientific computing concepts but it **is not** an applied module, i.e it will not explicitly spell out how to do certain operations. We therefore hope that this chapter will provide a nice introduction to implementing some of these operations.

As with so much else we have shown you, we don't write out these operations by hand, there are far more capable and knowledgeable people out in the world who have developed packages to perform scientific operations much faster than we ever could. For this we will not only use `numpy`, which you had an introduction to last chapter, but we will also focus on `scipy` which (unsurprisingly) stands for scientific python. This module is somewhat like `numpy`, in that it contains many sub modules for performing many different operations. These include integration, curve fitting, interpolation, signal processing and spatial computations to name a few.

10.2 Opening Different File Types

Regardless of what field you end up specialising in it's certain you'll be handling data in a range of different formats. There are standard formats like text files, `pickle` files, `numpy` files, hdf5 files, fits files, and csv files (which you've already seen). There are also a plethora

of proprietary formats used in specifics fields for specific functions; for instance the Met Office use a data format modelled after HDF5 files called netCDF4 files which they use specifically for storing weather data.

In this section we will cover the simple formats which you will come across again and again. Hopefully this introduction, and the demonstrations it contains, will mean you will be unfazed when presented with a file to open for an assignment or project in the future.

10.2.1 `with` and the importance of closing files

One thing which has been seen time and time again is files becoming corrupted due to careless handling. This isn't a big deal for a file which is backed up or can easily be reproduced but in many instances you might be working with a one off file containing terabytes of data which took days, if not weeks or months, to produce. It is therefore very important to learn how to correctly handle data files. The most important thing to remember is if you have opened a file you must close it! Many data files will allow you to do this manually in some way with a function called close. For instance in HDF5 (which we will not cover in great detail here) you can open and close a file like this:

```
In [1]: # Open a file
        hdf = h5py.File('myfile.hdf5', 'r')

        # Do some stuff with the contents of the file
        do_stuff(hdf)

        hdf.close()
```

Doing this safely closes the file, preventing possible corruption from closing improperly. Don't worry about the specifics of the above example, we include it simply as a demonstration of the different methods to open and close files.

The above method is implemented in a number of modules for handling different file formats but how it works is entirely format specific, and also specific to the module being used to handle that format. Thankfully `Python` includes the `with` statement structure. This statement will open a file and perform the operations indented below the `with` declaration before safely closing the file. This is structured in the following manner.

```
In [2]: with file as a:
            do_stuff(a)
```

You will see in the following sections the different things that 'file' can be replaced with. An important thing to note is that if you simply want to extract the data within a file you can simply assign it to a variable within the `with` statement and the data will be assigned to the variable even after the file has been closed.

```
In [3]: with file as a:  # open file
            x = a.read()  # store data in variable

        # File is closed

        # Perform operations on the data that was stored in file
        do_stuff(x)
```

Indeed (for completeness) you can combine the HDF5 example with "with" and can then omit the `.close()` call.

```
In [4]: with h5py.File('myfile.hdf5', 'r') as a:  # open file
            do_stuff(a)

        # File is closed
```

The following sections contain demonstrations of how to open different file types. The examples in these sections will all give errors if you try to run them, since you do not have the files quoted within them. They are simply provided here for reference.

10.2.2 Text and binary files

With simple text or binary files (these are files which aren't human readable but are saved with strings of binary or hex) we can use the `Python` open function combined with the `with` statement. The open function takes a file path as the first argument and a "mode" as the second argument. This mode can be `'r'` to read a file, `'w'` to write a file, `'a'` to append to an existing file or `'x'` to create a file, the last of these will error if the file already exists. To read a binary file you simply include `'b'` in the mode, i.e. `'rb'` to read a binary file.

```
In [5]: with open('myfile.txt', 'r') as a:
            x = a.read()
```

10.2.3 Pickle files

If we have a dictionary and we want to save it the best option is to use a pickle file. This format "serialises" the data and can be very efficient when it comes to saving large arrays, not just dictionaries. The extension for a pickle file can effectively be anything (the same is true for binary files) but convention is to use `.pkl` or `.pck` which we adopt here.

To use pickle files we need to import the `pickle` module and call it's `load` function while using `open` in binary mode.

```
In [6]: import pickle

        with open('myfile.pck', 'rb') as a:  # open file
            x = pickle.load(a)  # store data in variable
```

To save a pickle file we simply exchange `'rb'` for `'wb'`, `load` for `dump` and provide the variable for saving as the first argument to `dump` and `a` as the second.

```
In [7]: import pickle

        with open('myfile.pck', 'wb') as a:  # create file object
            pickle.dump(x, a)  # store data in file
```

10.2.4 `Numpy` files

`Numpy` also provides functions to save and load arrays. These functions do this in nice neat single line statements. In addition to the `numpy` specific format (`.npy`) it also has an interface for text files but do be aware that text files will be loaded containing strings rather than floats or integers.

```
In [8]: import numpy as np

        # Load a numpy file
        x = np.load('mynumpyfile.npy')

        # Load a text file with numpy
        x = np.loadtxt('mytextfile.txt')

        # Save a numpy file
        np.save('mynumpyfile.npy', y)

        # Save a text file with numpy
        np.savetxt('mytextfile.txt', y)
```

10.2.5 FITS files

FITS (Flexible Image Transport System) files are widely used in astronomy to store images and tabular data. They have a header which contains metadata about the images or tables. The `astropy` package contains a function for handling these files.

```
In [9]: from astropy.io import fits

        # Open the fits file
        hdu = fits.open('myimg.fits')

        # Get the first image in the file
        img = hdu[0].data

        # Close the file
        hdu.close()
```

A detailed look at this module and file format can be found here: `https://docs.astropy.org/en/stable/io/fits/`.
FITS tables can be read directly into `astropy` tables, which offer very similar functionality to pandas data frames, see https://docs.astropy.org/en/stable/table/index.html.

10.2.6 HDF5 files

HDF5 files are becoming the norm for storing large amounts of data. They are very time and memory efficient allowing a user to access huge datasets without waiting for the entire file to load. They are structured exactly like dictionaries with keys representing "groups" with datasets and attributes stored within groups. These groups can then also be nested just like a dictionary. To access a HDF5 file we can use the `h5py` module.

```
In[10]: import h5py

        # Open the HDF5 file
        hdf = h5py.File('myfile.hdf5', 'r')

        # Open 2 groups
        grp1 = hdf["myfirstgroupkey"]
        grp2 = hdf["mysecondgroupkey"]

        # Extract a dataset from group 1
        dset = grp1["thefirstdataset"]

        # This can also be done from the root
        dset = hdf["myfirstgroupkey"]["thefirstdataset"]

        # Get an attribute from group 2
        at = grp2.attr["thefirstattribute"]

        # Again this can be done from the root group
        at = hdf["mysecondgroupkey"].attr["thefirstattribute"]

        # Close the file
        hdf.close()
```

More details on using `h5py` and the structure of HDF5 files can be found here: `http://docs.h5py.org/en/stable/quick.html`.

10.3 Treating Images As Arrays

In the simplest terms an image is a 2D array of values representing intensity in each pixel. Hopefully the parallel to a 2D `numpy` array here should immediately come to mind. This means we can manipulate, probe and produce images using many of the processes we have already covered.

10.3.1 Producing images

To work with images we first need a way to see them. This is where `matplotlib` comes to the rescue. In addition to the `plot` and `scatter` functions you have already seen, there is also a function for plotting images `imshow`. To use it you simply supply your image as an argument. Lets set up a random 2D array and plot it as an image.

```
In [1]: # Create an iamge of 1000**2 random values
        img = rng.uniform(1, 100, (1000, 1000))

        # Plot the image
        plt.imshow(img)
        plt.show()
```

The code above will produce the following image.

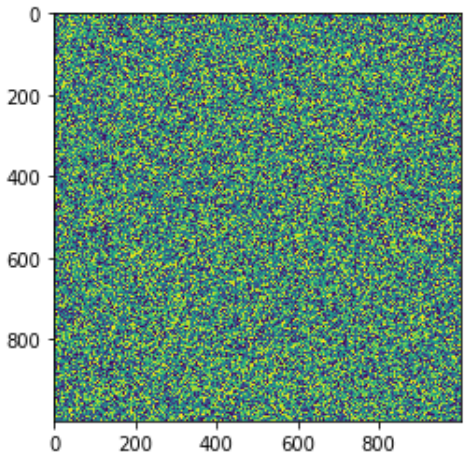


Figure 10.1: An image produced from a random uniform distribution.

Notice how `matplotlib` has attributed axis values to the image despite us giving it no information but the x and y ranges. What `matplotlib` has done is plot the number of pixels along each axis. Let's say we know that the extent of the x and y axes is 200 with the pixels centred on 0 in both axes. We can tell `matplotlib` this is the case using the `extent` keyword argument to `imshow`.

```
In [2]: # Plot the image with an extent
        plt.imshow(img, extent=[-100, 100, -100, 100])

        # Label the axes
        plt.xlabel(r'$x$ (arbitrary units)')
        plt.ylabel(r'$y$ (arbitrary units)')

        plt.show()
```

The list passed to the `extent` keyword argument is a list of the extremes along each axis of the form `[xmin, xmax, ymin, ymax]`. This will then produce the following image.

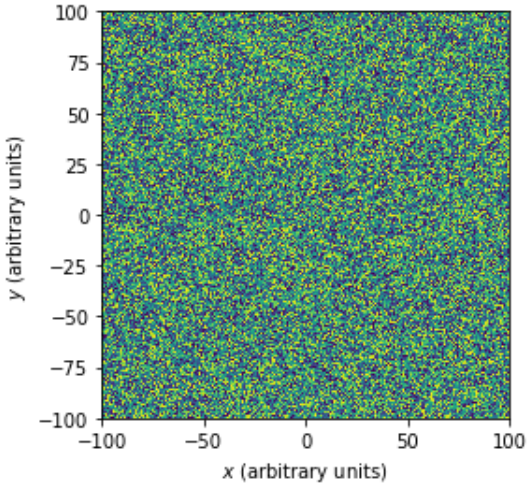


Figure 10.2: An image produced from a random uniform distribution showing the use of the `extent` keyword.

But what if we have an image where we don't want axis labels? In that case you can simply turn off the axes, although there are many ways to do this the simplest is to do the following.

```
In [3]: # Plot the image
        plt.imshow(img)

        # Remove the axes
        plt.axis(False)

        plt.show()
```

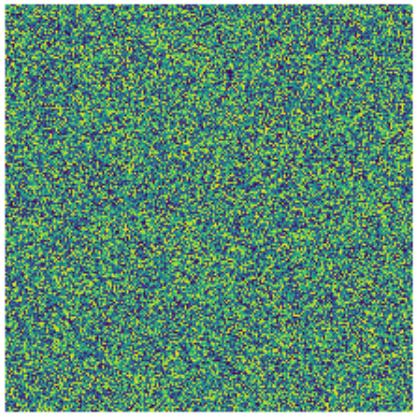


Figure 10.3: An image produced from a random uniform distribution with the axes removed.

The final thing to note is how to apply colormaps. These allow you to make your grayscale images (single values in pixels, rather than RGB arrays with 3 values in each pixel) "look prettier". A full list of the available colormaps can be found here: `https://matplotlib.org/tutorials/colors/colormaps.html`. To use a colormap you simply supply the name of that colormap to the `cmap` keyword argument.

```
In [4]: # Plot the image with the plasma colormap
        plt.imshow(img, cmap='plasma')

        # Remove the axes
        plt.axis(False)

        plt.show()
```

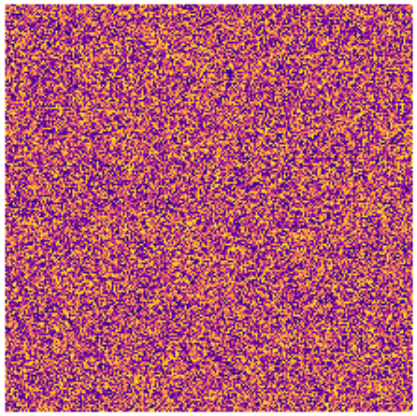


Figure 10.4: An image produced from a random uniform distribution with the plasma colormap.

10.3.2 Manipulating images

We can treat these images as simple arrays and perform any operation on them that you could on an array. Beyond this however there are a large number of image processing specific modules such as `PIL`, `openCV` and `scikit-image`. We won't go into those here but there's a plethora of image processing possibilities. Here we will just demonstrate some simple operations.

Lets make a more "interesting" image to do this, a 2D Gaussian. These can be used as an approximation for a poorly resolved point source, or star. To do this we can define a 2D Gaussian function and x and y ranges. From these we can calculate the image with a neat trick to avoid looping.

```
In [5]: # Define a 2D Gaussian function
        def gauss2d(x, y, mx=0, my=0, sigx=1, sigy=1):
            norm = 1. / (2. * np.pi * sigx * sigy)
            exponent = -((x - mx)**2. / (2. * sigx**2.)
                         + (y - my)**2. / (2. * sigy**2.))
            return norm * np.exp(exponent)

        # Define the x and y values
        xs = np.linspace(-5, 5, 1000)
        ys = np.linspace(-5, 5, 1000)

        # Combine the x and y ranges into a 2d grid (this is the neat trick)
        xx, yy = np.meshgrid(xs, ys)

        # Compute the gaussian image leaving the default values of 0 for the
        # means and 1 for the standard deviations
        img = gauss2d(xx, yy)

        # Plot the image with the known extent and inferno colormap
        plt.imshow(img, extent=[-5, 5, -5, 5], cmap='inferno')

        # Label axes
        plt.xlabel(r'$x$ (arbitrary units)')
        plt.ylabel(r'$y$ (arbitrary units)')

        plt.show()
```

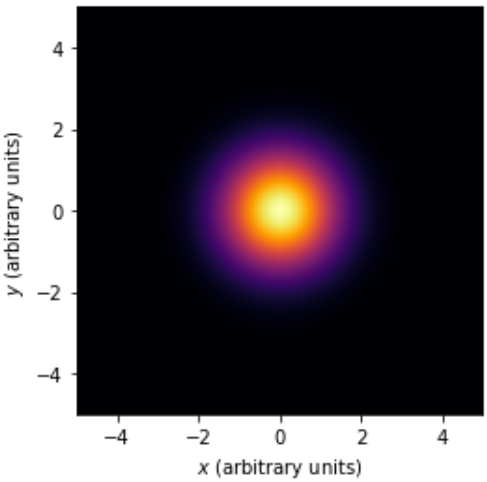


Figure 10.5: An image of a basic 2D Gaussian.

We can obviously do simple calculations on the image to scale it in different ways but you can play around with that yourself. A more useful function to apply is the base 10 logarithm, this can be very useful for bringing out detail in pictures where a bright set of pixels dominates the image.

```
In [6]: # Scale the image logarithmically
        logimg = np.log10(img)

        # Plot the image with the known extent and inferno colormap
        plt.imshow(logimg, extent=[-5, 5, -5, 5], cmap='inferno')

        # Label axes
        plt.xlabel(r'$x$ (arbitrary units)')
        plt.ylabel(r'$y$ (arbitrary units)')

        plt.show()
```

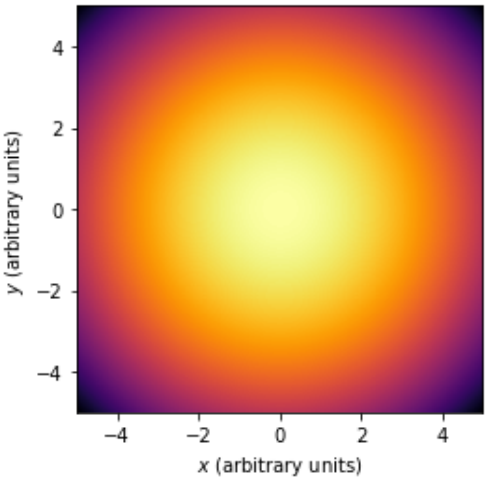


Figure 10.6: An image of a basic 2D Gaussian with logarithmic scaling.

We can also apply Boolean conditions to control specific regions in the image. Lets remove the background, to do this we can simply define an intensity threshold below which we set the pixel values to `np.nan`, `matplotlib` will then interpret these values as transparent.

```
In [7]: # Set any values below 10% of the maximum of the image to be
        # transparent
        trans_img = img.copy()
        trans_img[img < img.max() * 0.1] = np.nan

        # Plot the image with the known extent and inferno colormap
        plt.imshow(trans_img, extent=[-5, 5, -5, 5], cmap='inferno')

        # Label axes
        plt.xlabel(r'$x$ (arbitrary units)')
        plt.ylabel(r'$y$ (arbitrary units)')

        plt.show()
```

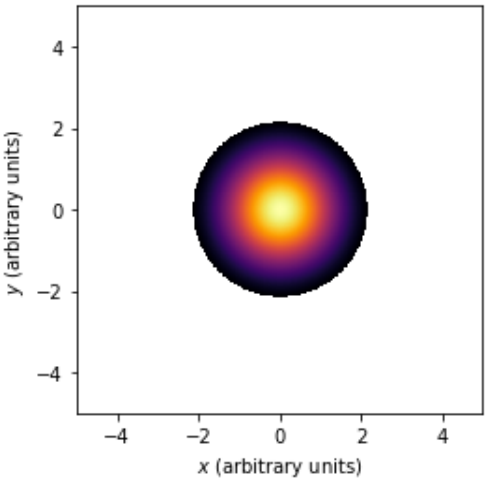


Figure 10.7: An image of a basic 2D Gaussian with a transparent background.

One final operation to consider, although it's not of much use here, is the clipping of an image between minimum and maximum intensity values. This will set any values below the minimum to the minimum and any values above the maximum to the maximum. This can be done a number of ways but the simplest is applying a minimum and maximum with `vmin` and `vmax`.

```
In [8]: # Plot the image with the known extent and inferno colormap,
        # setting minimum and maximum intensities
        plt.imshow(img, extent=[-5, 5, -5, 5], cmap='inferno',
                   vmin=img.max()*0.5, vmax=img.max()*0.8)

        # Label axes
        plt.xlabel(r'$x$ (arbitrary units)')
        plt.ylabel(r'$y$ (arbitrary units)')

        plt.show()
```

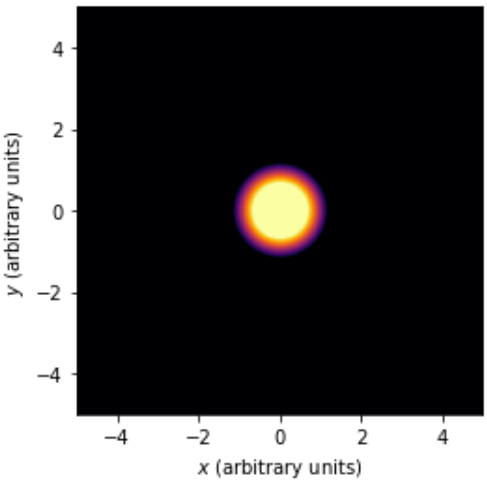


Figure 10.8: An image of a basic 2D Gaussian with intensities clipped to minimum and maximum values.

10.3.3 Exercises (with worked solutions)

We can now apply this to some exercises with a more interesting image.

1. Download from canvas and open the `pickle` file "Hubbleimg.pck" assign the contents to a variable. Plot the image contained in the file. You shouldn't see anything in the image as this is a raw image with minimal processing. This is an image of the pillars of creation taken using the Hubble Space Telescope, although you wouldn't know that at this point.

2. Try scaling the image with different mathematical functions to bring out detail and plot your tests. (Hint: log10 scaling is a good start but there are many many possibilities, have a play).

3. Use `vmin` and `vmax` to improve the quality of your image and bring out the maximum detail possible.

4. (No solution) Making these images look good is more art than science. Try to make the best image possible by applying anything you can think of to help. (Hint: You might be able to reduce noise by subtracting the mean or median and re-normalising the values, or maybe there's different scaling you could apply, maybe even look into `PIL` at the image processing possible with it).

10.4 Histograms

In previous exercises (mostly advanced exercises) we have asked you to create histograms as with a bit of brute force it's easy enough to code one up using dictionaries or lists. This said, they are a very powerful analysis tool and come up again and again when analysing data, in fact you should have come across them in labs. To make a histogram you need to sort data into a set of bins. In `Python` there are a number of ways to produce histograms, we will show the `numpy` method, there is nothing wrong with any other method and the `matplotlib` method is similar and produces a plot at the same time, `numpy` just provides a bit more flexibility before plotting.

To produce a histogram we first need some data to histogram, for this we can just produce random values from a normal distribution, and we need bins in which to sort it. To begin with we can just tell `numpy` that we want a certain number of bins, a good rule of thumb if you don't explicitly know how many bins you need is to use the square root of the number of data points you have.

```
In [2]: # Define an array of values to histogram
        vals = rng.normal(0, 10, 10000)

        # Compute the number of bins (this must be an integer)
        nbin = int(np.sqrt(vals.size))

        # Compute the histogram
        H, bin_edges = np.histogram(vals, bins=nbin)
```

We now have the values sorted into bins where `H` is the number of counts in each bin and `bin_edges` is, unsurprisingly, the edges of the bins. These variable names are convention, you can obviously change these. Lets plot these results as a bar graph using `bar`.

```
In [2]: # Plot the histogram as a bar graph
        plt.bar(bin_edges, H)
        plt.show()
```

Of course, you may have seen this coming... but we have an error, specifically "`ValueError: shape mismatch: objects cannot be broadcast to a single shape`'. This is because `H` and `bin_edges` are different sizes because `bin_edges` contains the edges of the bins, i.e. 1 more entry than `H`. We want to plot against the centre of each bin. Not to worry though, this is easily achieved.

```
In [2]: # Compute the bin centres
        bin_width = bin_edges[1] - bin_edges[0]
        bin_cents = bin_edges[1:] - (bin_width / 2)

        # Plot the histogram as a bar graph
        plt.bar(bin_cents, H)

        # Label axes
        plt.xlabel(r'$x$')
        plt.ylabel(r'$N$')

        plt.show()
```

Running the above code produces the plot below (although yours will look subtly different due to the nature of unseeded random numbers). Notice how simply by histograming the values we have roughly reproduced the normal distribution the values were pulled from.

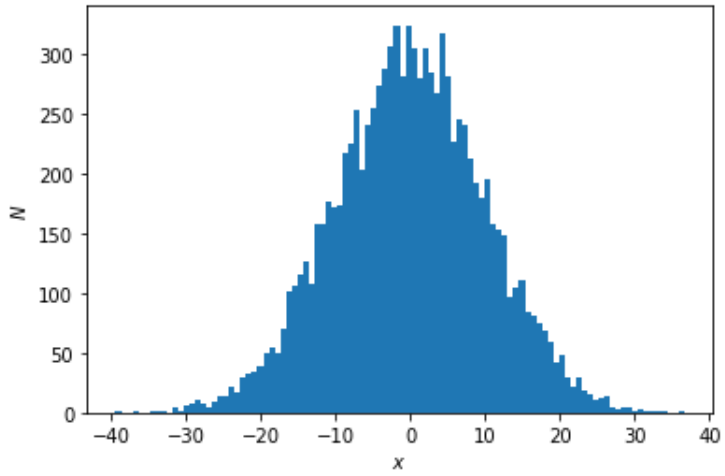


Figure 10.9: An example histogram of values pulled from a normal distribution.

Another way to achieve the same result is by defining the bins ahead of time, this gives the histogram an explict range.

```
In [2]: # Define an array of values to histogram
        vals = rng.normal(0, 10, 10000)

        # Compute the number of bins (this must be an integer)
        nbin = int(np.sqrt(vals.size))
        bins = np.linspace(-40, 40, nbin)

        # Compute the histogram
        H, bin_edges = np.histogram(vals, bins=bins)
```

We can also produce 2D histograms which are themselves just images, think of a camera collecting photons (counts) in pixels (bins). We can approximate the Gaussian distribution from Sec.10.3.2 very simply by creating a 2D histogram in the same way we just produced a 1-dimensional histogram. Here we will set the number of bins as we did originally (this will apply along both axes) and also define a range for the bins to cover, this is similar to providing an array of bins but instead defines the range in which you put `nbin` bins, in a similar manner to defining the extent of an image in `imshow`.

```
In [2]: # Define the x and y values from a normal distribution
        xs = rng.normal(0, 1, 100000)
        ys = rng.normal(0, 1, 100000)

        # Define the number of bins
        nbin = 100

        # Define the range for the bins
        binrange = ((-5, 5), (-5, 5))

        # Compute the histogram using the range
        H, xbin_edges, ybin_edges = np.histogram2d(xs, ys, bins=nbin,
                                                   range=binrange)

        # Plot the image
        plt.imshow(H, cmap='viridis', extent=[-5, 5, -5, 5])

        # Label axes
        plt.xlabel(r'$x$')
        plt.ylabel(r'$y$')

        plt.show()
```

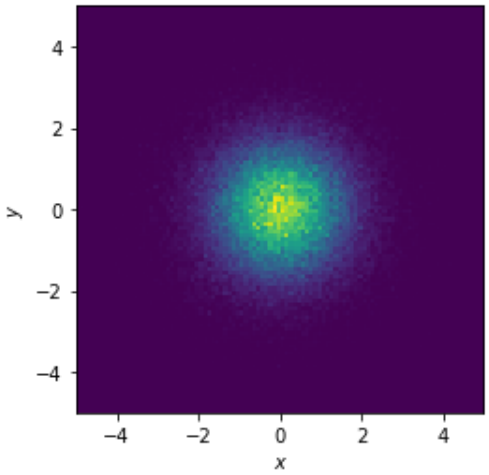


Figure 10.10: A 2D Gaussian distribution produced using a 2D histogram.

This can be very useful for creating images from 1D spatial data.

10.5 Basic Curve Fitting

As you no doubt have already seen, when working with scientific data you're often fitting your experimental data to a model or theory. You can then compute statistics to determine the validity of any theories you may be comparing to. Although you can do this in excel with `LINEST`, this is very limited both in implementation and in the logistics of working with large data sets. Instead we want to utilise `scipy.optimize` which contains a fitting function called `curve_fit`. Here we will demonstrate how to use `curve_fit` so that your ready when you need to use it.

First download the file `curvefitting.txt` from canvas. We will load this data into your notebook and then fit it with a straight line. To use `curve_fit` we need 3 things, the data, a function with the first argument being the x values and any other parameters as the subsequent arguments, and an initial guesses for each of the parameters. Once we have these set up we can call `curve_fit` and use the produced parameters to plot the fit.

```
In [2]: from scipy.optimize import curve_fit

        # Load the data
        arr = np.loadtxt('curvefitting.txt')

        # Convert the data from strings to floats
        arr = np.float64(arr)

        # Extract the xs and ys from the file
        xs = arr[:, 0]
        ys = arr[:, 1]

        # Define a function to fit the data to
        def func(x, m, c):
            return m * x + c

        # Define some initial guesses for m and c
        guesses = [1, 1]

        # Get the fit
        popt, pcov = curve_fit(func, xs, ys, p0=guesses)

        # Compute the fit using the parameters in popt
        fit = func(xs, popt[0], popt[1])

        # Plot the resulting fit and data
        plt.scatter(xs, ys, marker='+')
        plt.plot(xs, fit, linestyle='--')

        # Label axes
        plt.xlabel(r'$x$')
        plt.ylabel(r'$y$')

        plt.show()
```

There are a few things to note in the code snippet above. Firstly, when opening a text file the contents within will have been converted to strings, to deal with this we have used `np.float64` to convert the array into 64-bit floats. Secondly, the data is stored in a (100, 2) array with the xs in the first column and the ys in the second column, we have had to use slicing to extract these into their own arrays.

On the topic of the guesses, `curve_fit` is fairly capable of handling even bad guesses so anything in the ball park of the right values should suffice. If you provide guesses miles from the truth it is possible for the fitting to fail, try it and see what happens. Finally, take a look at what is returned by `curve_fit`, `popt` contains the optimal parameters in the same order they appear in the arguments of the function, `pcov` on the other hand is the covariance matrix, this is a concept from statistics which you will come across in the future so we will skirt around the complexities but it is enough right now to say the diagonal elements of `pcov` are the uncertainties on the returned parameters.

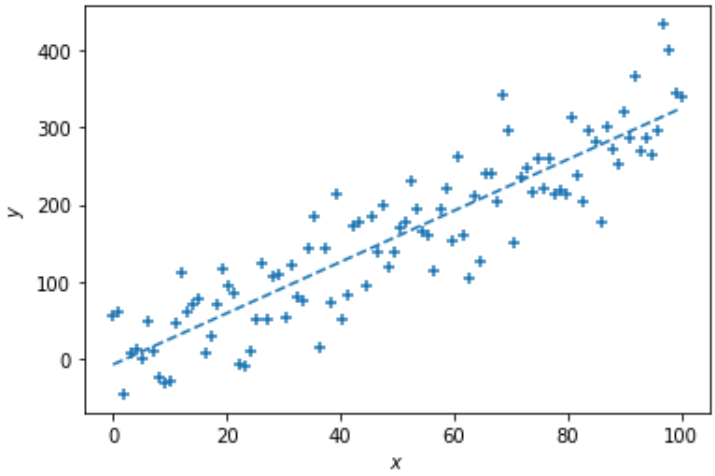


Figure 10.11: A plot of the fit produced using `curve_fit`.

Hopefully this serves as a brief introduction to using `curve_fit` and it won't cause you distress when you come across it in the future. We will cover another example in the exercises.

10.6 Integration

Another staple in scientific computing is the numerical computation of integrals, with some integrals not having analytical solutions at all. You may have come across ways to approximate integrals in the past such as the trapezium and Simpson's rules. There are multiple implementations of these from different modules, here we will cover `scipy` but do note that these functions can be used in a number of ways, either providing x values or an interval dx at which to calculate the integration.

To compute an integral with these functions we will define two arrays one of x values and one containing the result from a 1D Gaussian, normalised such that the integral should be unity. We can then apply the functions and see if we sampled with fine enough resolution to get the expected result.

```
In [2]: from scipy.integrate import simps, trapz

        # Define some xs
        xs = np.linspace(-50, 50, 1000)

        # Define a 1D Gaussian function
        def gauss(x, mx=0, sigx=1):
            norm = 1. / np.sqrt(2. * np.pi * sigx**2)
            exponent = -(x - mx)**2. / (2. * sigx**2.)
            return norm * np.exp(exponent)

        # Compute the Gaussian
        ys = gauss(xs)

        # Compute the integral by the trapezium and Simpson's rule
        trap = trapz(ys, xs)
        simp = simps(ys, xs)

        print(trap, simp)
```

```
        1.0 1.0
```

Evidently in both case we did sample with high enough resolution and over a large enough range. That said, for such a simple function this shouldn't be a surprise.

In addition to the trapezium and Simpson's rules `scipy` also contains a function called `quad` which allows you to compute integrals from a function. Let's reproduce the above example but using `quad`. To use `quad` we need to define a 1D Gaussian as a function and define upper (b) and lower (a) bounds for the integration.

```
In [2]: from scipy.integrate import quad

        # Define some xs
        xs = np.linspace(-5, 5, 1000)

        # Define a 1D Gaussian function
        def gauss(x, mx=0, sigx=1):
            norm = 1. / np.sqrt(2. * np.pi * sigx**2)
            exponent = -(x - mx)**2. / (2. * sigx**2.)
            return norm * np.exp(exponent)

        # Compute the integral by the trapezium and simpsons rule
        qu = quad(gauss, a=-50, b=50)

        print(qu)
```

```
        (1.0000000000000002, 1.0346447325665705e-12)
```

At first glance `quad` didn't give as perfect a result as the simpler methods above, however one nice thing about `quad` is it also computes an error on the integration. Taking this error into account the result is indeed consistent with the expected result of unity. The inclusion of an error makes `quad` very useful and as functions get more complex `quad` really begins to shine.

We have not covered the entirety of the possible integration methods provided by `scipy`, for a full list and a tutorial see `https://docs.scipy.org/doc/scipy/reference/tutorial/integrate.html`.

10.6.1 Exercises (with worked solutions)

1. Generate the 1D Gaussian from the examples in the histogram section but this time taking only 1000 values.

2. Create a histogram of the random samples.

3. Define a Gaussian function and fit it. Plot the resulting histogram and fit as a bar plot and dashed line respectively. (Hint: you can make the normalisation a fitting parameter as well).

4. Integrate $\sin^2(x)\cos^3(x)$ and $\sin^2(x)\cos^4(x)$ between -2π and 2π. Before you calculate them do you know immediately which should be 0? There's a hidden integration trick here with multiplying odd and even functions when combined with symmetric limits.

10.7 Worked Solutions

10.3.3

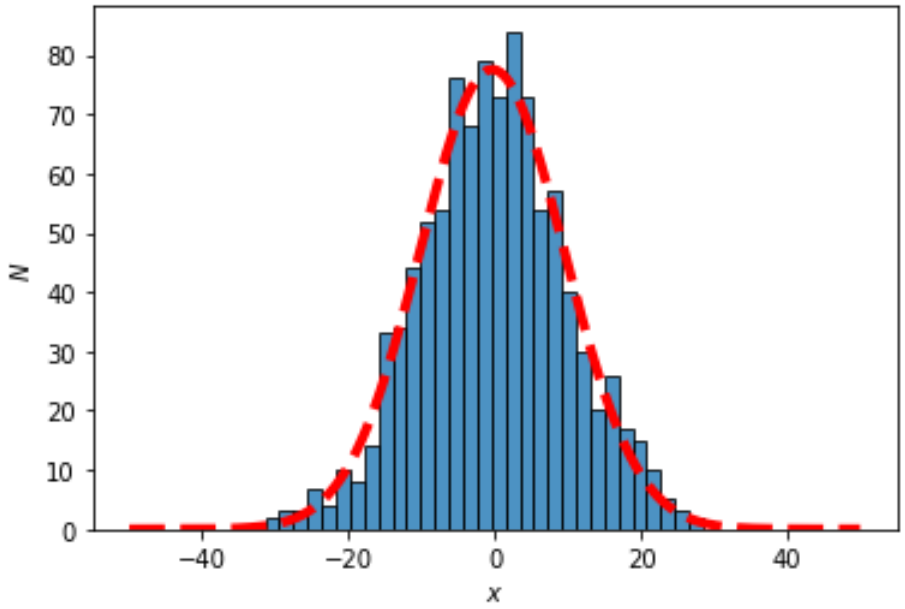
```
In [1]: import pickle
        import matplotlib.pyplot as plt
        import numpy as np

        # Question 1

        with open("Hubbleimg.pck", 'rb')as pfile:
            img = pickle.load(pfile)

        # Plot raw image
        plt.imshow(img, cmap='Greys_r')
        plt.axis(False)
        plt.show()
```

```
In [2]: # Question 2
        # Scale the image to better show detail
        logimg = np.log10(img)
        asinimg = np.arcsinh(img)

        # Plot each image
        plt.imshow(logimg, cmap='Greys_r')
        # NOTE: log10 produces some -inf where there are 0 counts,
        # hence the error
        plt.title("Log scaling")
        plt.axis(False)
        plt.show()
        plt.imshow(asinimg, cmap='Greys_r')
        plt.title("Arcsinh scaling")
        plt.axis(False)
        plt.show()
```

```
In [3]: # Question 3
        # Find minima and maxima counts that correspond to detail in
        # the nebula
        print(np.min(logimg[np.logical_and(logimg != -np.inf,
                                           ~np.isnan(logimg))]))
        print(np.min(asinimg))
        print(np.max(logimg[~np.isnan(logimg)]))
        print(np.max(asinimg))

        # Plot the image setting the minimum value and maximum
        plt.imshow(logimg, cmap='Greys_r', vmin=1.5, vmax=0.5)
        plt.title("Log scaling Clipped")
        plt.axis(False)
        plt.show()
        plt.imshow(asinimg, cmap='Greys_r', vmin=-0.1, vmax=0.75)
        plt.title("Arcsinh scaling Clipped")
        plt.axis(False)
        plt.show()

        # We can permenantly apply these changes to the images using np.clip
        logimg = np.clip(logimg, a_min=1.5, a_max=0.5)
        asinimg = np.clip(asinimg, a_min=-0.1, a_max=0.75)

        # NOTE: these values aren't "correct" (neither are the choices of
        # scalings the only possible choices) they are the values chosen
        # while doing this worked example. There almost certainly are
        # better combinations!
```

10.6.1

```
In [1]: import numpy as np
        import matplotlib.pyplot as plt
        from scipy.integrate import quad
        from scipy.optimize import curve_fit

        # Question 1
        # Define an array of values to histogram
        vals = rng.normal(0, 10, 1000)


        # Question 2
        # Compute the number of bins
        nbin = int(np.sqrt(vals.size))

        # Compute the histogram
        H, bin_edges = np.histogram(vals, bins=nbin)

        # Question 3
        # Define a 1D Gaussian function
        def gauss(x, norm, mx, sigx):
            exponent = -(x - mx)**2. / (2. * sigx**2.)
            return norm * np.exp(exponent)

        # Define guesses for the gaussian parameters
        guesses = [100, 0, 10]

        # Compute the bin centres
        bin_width = bin_edges[1] - bin_edges[0]
        bin_cents = bin_edges[1:] - (bin_width / 2)

        # Get the fitting parameters
        popt, pcov = curve_fit(gauss, bin_cents, H, p0=guesses)

        print('Fit parameters are:', popt)

        # Evaluate the fit
        fit_xs = np.linspace(-50, 50, 10000)
        fit = gauss(fit_xs, popt[0], popt[1], popt[2])

        # Plot the histogram as a bar graph
        plt.bar(bin_cents, H, width=bin_width, alpha=0.8, edgecolor='k')
        plt.plot(fit_xs, fit, linestyle='--', color='r', linewidth=4)

        # Label axes
        plt.xlabel(r'$x$')
        plt.ylabel(r'$N$')

        plt.show()
```




```
In [2]: # Question 4

        # It's np.sin(x)**2*np.cos(x)**3 which should be 0 as this is an
        # odd function (cos^3) multiplied by an even function (sin^2)
        # which results in an odd function, the integral of any odd function
        # over limits symmetric about 0 is NECESSARILY 0!

        # Evaluate integrals defining functions as lambda functions
        # for neatness
        qu1 = quad(lambda x: np.sin(x)**2*np.cos(x)**3, -2*np.pi, 2*np.pi)
        qu2 = quad(lambda x: np.sin(x)**2*np.cos(x)**4, -2*np.pi, 2*np.pi)

        # Print result
        print(qu1)
        print(qu2)
```