

# PYTHON PROGRAMMING

## Top Of The Stack

```
1  # %% Hello World Program
2
3  print('Hello World')
4
5  |
```

DARREN LEFCOE

# Python Programming

Top of the stack

## Introduction

How to improve your ranking, become a better Python programmer and a lateral thinker.

Learn core concepts and the new practices of problem solving. Harness the immense community knowledge whilst challenging the errors with expertise. We explore common questions and answers that any coder will invariably encounter.

Written by scientists and programmers who rank in the top 1% of contributors on the stackoverflow website and have consistently maintained this position for a significant period of time. This book demonstrates how you can achieve the same.

Some of the contributions are *work related* and others are simply *out of curiosity*.

**Nevertheless, these are the questions, answers and techniques that will take you to the top of the stack.**

The community embraces you and will enjoy your feedback and contributions.

## About the author

He holds a Master's degree in Physics from Imperial College, London and has over a quarter of a century of experience in the engineering and banking and finance industry, mostly as a quantitative trader. He has worked in startup and mid sized companies and also multinational organisations.

In recent years, his focus has been on Mathematics and Python programming which is a popular language for data science, machine learning and artificial intelligence.

He has engaged with many of the top contributors in the coding and science community and brings this wealth of knowledge to the reader.

Always at the forefront of technology and a driver of change to improve. He now shares some of the knowledge that he has acquired over this journey that will help you in yours by accelerating your understanding and progress.

The baton passes from one hand to the next...

## Acknowledgements

Firstly, with eternal gratitude to my mother who, whilst frail and small of frame, at 80 years of age has proved to be a rock of the family. With a heart of gold and a mentality of steel, this kind lady who was a refugee from the Middle East, undertook the hardest life along with her now deceased but honourable and loving husband to selflessly put three children through private school. They instilled upon their children the gift of discipline, education and hard work and never has this been so appreciated in the

writing of such. It is not even possible, even if one wanted, to express enough gratitude towards them.

To brother and sister and their children. You have always been the one constant in the life of the writer. Having shared the good times and the bad times with equal pleasure and drive and long may this continue. It is your strength and happiness that keeps those around you going and gives your sibling the passion for what he does.

And finally, to my three beautiful daughters. You are the stars of my life and represent the future generation. May you continue your journey for us all with strength, endeavour and happiness.

## Table of contents

[Introduction](#)

[About the author](#)

[Acknowledgements](#)

[Table of contents](#)

[A brief introduction to python](#)

[Background](#)

[Nomenclature](#)

[Structure](#)

[What is the stack?](#)

[Who uses the stack?](#)

[What is a stack overflow?](#)

[Let's get started](#)

[The code](#)

[The editor](#)

## Python Basics

Hello world

What is available

Comments

The data types

Variables

Lists, Dictionaries and Tuples

How to count

Mutable and immutable

Basic algebra

For and while loops

If else logic

Functions

Classes

Magic methods

Importing modules

Logical order

Making modules

The main module

## Coding Techniques

What is a coding problem

What is a coding technique

Recursive functions

Backtracking

list comprehension

Binary Tree

## Fun problems

Perfect numbers

[Swap apples for ideas](#)

[Rescue boats](#)

[Bracket Matching](#)

[Collatz Conjecture](#)

## [Data Science](#)

[Understanding Data](#)

[Data exploration: beginner](#)

[Data exploration: intermediate](#)

[Data visualisation](#)

[Machine learning](#)

## [Finance](#)

[Time series](#)

[Cleaning data](#)

[Moving averages](#)

[Reading data](#)

[Cosine similarity](#)

## [The web](#)

[HTTP request](#)

[Websockets](#)

[Asynchronous](#)

## [Graphical user interfaces](#)

[Hello world GUI](#)

[Password entry](#)

[Animations](#)

## [Maths](#)

[Fermat's last theorem](#)

[Complex numbers](#)

[Parametric equations](#)

[Derivatives](#)

[Maxima, minima & points of inflexion](#)

[Newton raphson](#)

[Graph a square](#)

[Dynamic systems](#)

[Infectious diseases](#)

[Rumour dynamics](#)

[Fractals](#)

[Mandelbrot set](#)

[Double pendulum](#)

[Three body problem](#)

[Predator vs prey cycles](#)

[Conway Game of life](#)

[The Lorenz Attractor](#)

[Digital Image Processing](#)

[Introduction to images](#)

[Reading images and videos](#)

[Basic functions](#)

[Resizing and cropping](#)

[Shapes and text](#)

[Warp perspective](#)

[Colour detection](#)

[Shape detection](#)

[Facial recognition](#)

[Virtual painting](#)

[Web scraping](#)

[Selenium](#)

[Beautiful soup](#)



[SQLite \(Database\)](#)

[Automation processes](#)

[Others](#)

[My first problem](#)

[My first answer](#)

[100 prisoners](#)

[Classes and mutability](#)

## A brief introduction to python

Python is a programming language that lets you work quickly and integrate systems more effectively.

Python is open source and free to use for anybody that has a computer and wants to learn.

A formal description from Wikipedia reads like this:

*“Python is a high-level, interpreted, general-purpose programming language. Its design philosophy emphasises code readability with the use of significant indentation”.*

So let's break this down:

<b>Term</b>	<b>What it means</b>
High-level	A bit easier for humans to read
Interpreted	Runs as it is written
General-purpose	A variety of use cases
Indentation	Uses indents instead of {}

## Background

The language itself was released in the early 90's. Then around ten years later a new version which introduced new features came out (version 2.0) and finally about 8 years after this a newer version came out (version 3.0).

Version 3.0 is not completely backward compatible with version 2.0 which means that some of the 2.0 code will not work with the 3.0 version. Most notably, the print statement which gained a set of brackets in the newer version.

Version 2.0: print "hello world"

Version 3.0: print ("hello world")

The backward incompatibility can sometimes be annoying and was considered to be a blunder by some whereas others consider it necessary. In particular, version 2.0 was sunset, which means it is *no longer supported*, as of Jan 2020.

So as time progresses the 3.0 code becomes the **main accepted version** and this is what this book focuses on.

When we talk about Python we talk about Python 3.0 .

Over this time, Python has consistently ranked as one of the top (most popular) programming languages and in recent years has made a resurgence due to use cases in the data science field primarily due to an abundance of modules written by a very competent community of users.

## Nomenclature

It is invariably necessary to go through some of the terms and phrases that are commonly talked about.

### **Pythonic:**

Code that is written in python looks both different and similar to other code. There are also new features (such as list comprehension in place of for loops) that other languages do not have and using those is considered to be Pythonic.

## Pythonista:

In short, this is someone who uses the python programming language. Someone who writes in a Pythonic way, which is considered to be concise and clean, might consider themselves to be a Pythonista.

## Structure

There are probably three main features that stand out from other predecessor languages and those are:

1. White space replacing curly braces `{}`
2. No semicolons `;` at the end of lines
3. Commented lines use `#` in place of `//`.

## White space vs indentation:

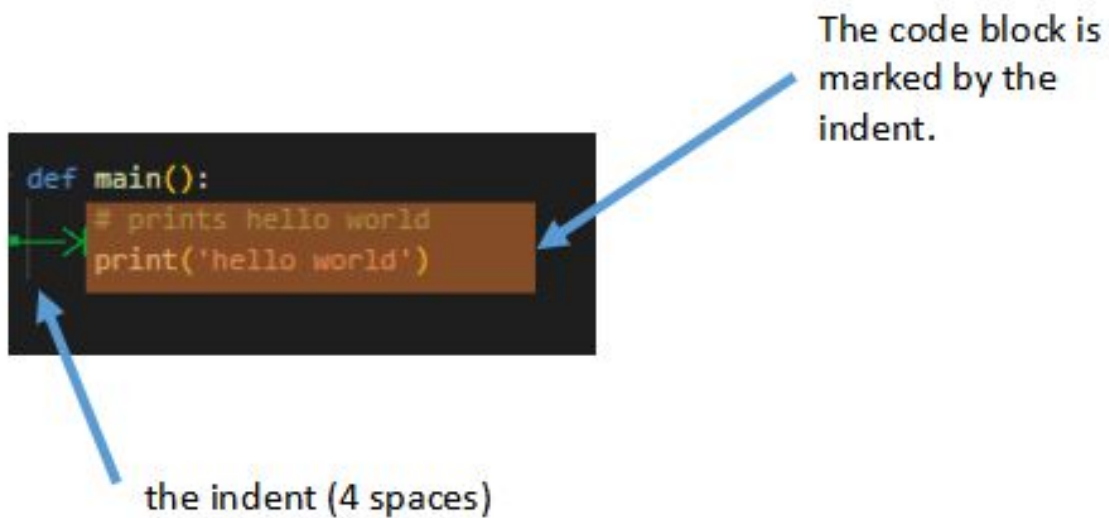
The most notable feature of this is the use of white space and indents compared to curly braces `{}` in many other languages.

All code to date is written in blocks and the reader has been able to identify the code blocks with the curly braces. This is one of the key introductions of python that was unique to its predecessors and in particular at the time the `c` and `c++` languages.

```
int main()
{
    // prints hello world
    printf("Hello World");
    return 0;
}
```

The code block is  
between the  
curly braces

Whereas the python version looks like this:



### Semicolon line ends:

The reader will also note the semicolon line ends of many languages. Python does not have this by design as it is intended to be cleaner.

Lines end in python at the end of the line, so the semicolon becomes unnecessary.

### A comment is # compared to // :

Finally, we see that single line comments in python use # in place of // which are common in other languages. There is no particular reason for this (after extensive google search) so presumably it is aesthetic as well.

There are many other differences, but as this is a basic overview they will be left to later on to cover those.

## What is the stack?

A good formal description comes from Wikipedia:

*“Stack Overflow is a question and answer website for professional and enthusiast programmers. It is the flagship site of the Stack Exchange*

*Network. It was created in 2008. It features questions and answers on a wide range of topics in computer programming*". There are roughly 14 million users registered on the platform.

The stack rewards and punishes for contribution with up or down votes for positive and negative contributions respectively.

Or more pythonically put:

The stack **rewards** and **punishes** for contribution:

With **up** or **down** votes

For **positive** and **negative** contributions

Respectively.

The basic format is this:

Good question	Up vote
Good answer	Up vote
Good comment	Up vote
Bad question	Down vote
Bad answer	Down vote

The above voting system, whilst sometimes harsh for the beginner, keeps the community honest by rewarding contributions and punishing distractions.

Just like any other platform or system, there are specific nuances and ways of asking and answering questions. The stack is quite direct with “a Minimal, Reproducible Example” approach.

This basically means that good questions isolate very specific problems in a “short and concise” manner and helps target solutions quickly. Likewise, the same is for the answers, “short and concise” wins.

So how does one get to the top of the stack?

Quite simple really, with a thirst to learn and contribute and by being a “good citizen”.

### Who uses the stack?

Basically anybody who has ever asked a programming question in any language has probably visited the stackoverflow website.

The most recent statistics show that there are around 50 million users and about half of those are professional developers and university students. So the other half are enthusiasts or novices or some other cohort of people with an interest in coding.

There are plenty of other excellent resources and websites and also, as for any programming language, the documents are excellent.

Nevertheless, sometimes it is simply more efficient to look up an answer on the web. We often “google it” and invariably stackoverflow with its plethora of completeness (for a whole variety of programming languages) will rank in or among the top few links.

## What is a stack overflow?

A stack overflow is a type of buffer overflow error that occurs when a computer program tries to use more memory space in the call stack than has been allocated to that stack.

One of the most common causes of a stack overflow is the recursive function, which is a type of function that repeatedly calls itself in an attempt to carry out specific logic. Each time the function calls itself, it uses up more of the stack memory. If the function runs too many times it consumes all the available memory, resulting in a stack overflow. This invariably leads to corruption of variables and data and is likely to cause the program to crash.

The term in many ways is suitable for the name of the community website as this is one of the problems that many of the users are trying to avoid.

The users go to [stackoverflow](https://stackoverflow.com) to find out how to prevent a stack overflow.



## Let's get started

The journey started long before Python, but the procedure for any language is the same. Create a `hello world` program. From there we can do almost anything.

### The code

Each language will have a source and place to download the code. And for Python this is the same in this regard. All the user has to do in today's world is to "google" the word python. This takes the user to the main website [<https://www.python.org>] and off they go.

Just click  
here

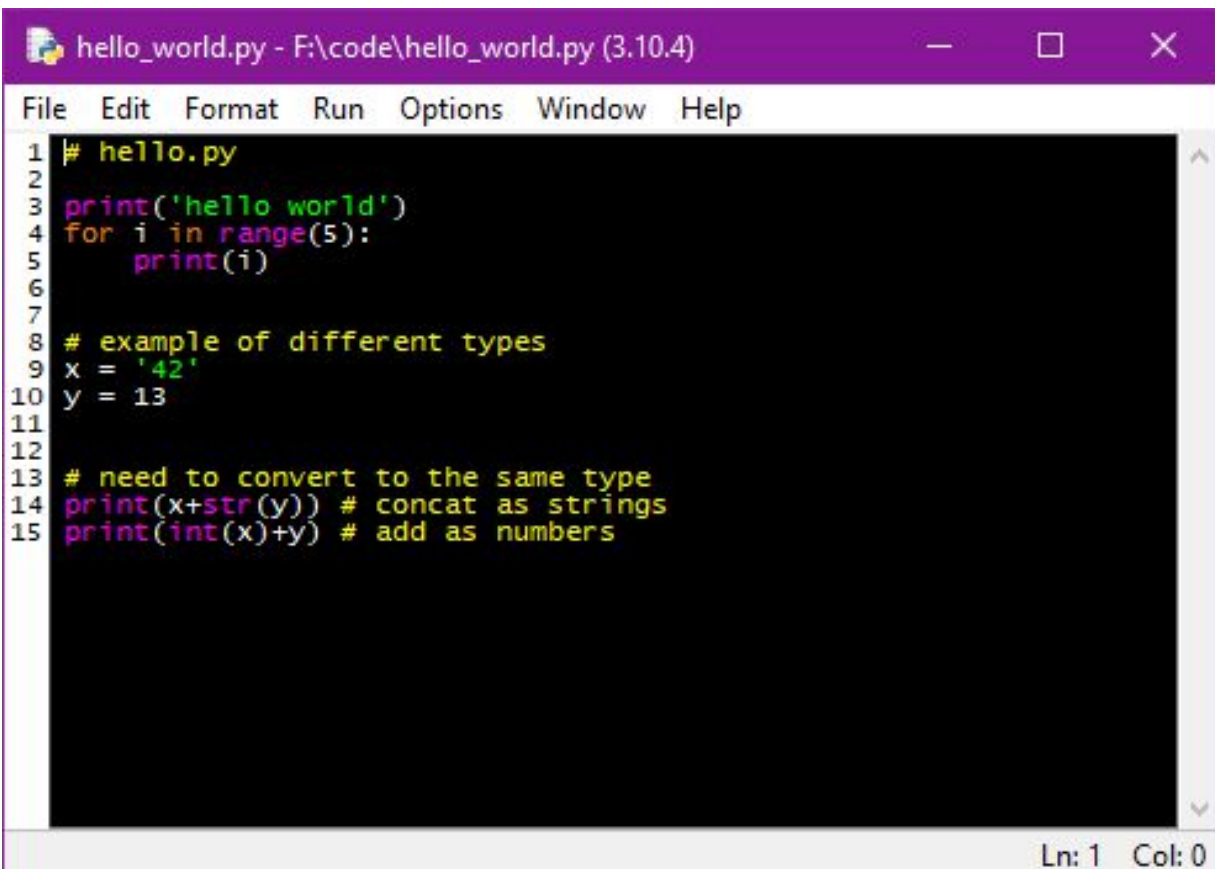


The user may wish to navigate to the appropriate version for the different operating systems (Windows, MacOS or Linux), but it is basically as simple as that.

### The editor

As part of the download for python, the user gets the default package editor (called IDLE) which is perfectly fine to use but lacks some of the main features that other editors have.

The IDLE stands for **I**ntegrated **D**evelopment and **L**earning **E**nvironment. It is an integrated development environment (IDE) for Python and it basically looks like this:

A screenshot of the IDLE Python IDE. The window title is 'hello\_world.py - F:\code\hello\_world.py (3.10.4)'. The menu bar includes 'File', 'Edit', 'Format', 'Run', 'Options', 'Window', and 'Help'. The code editor has a black background with syntax-highlighted Python code. The code includes a 'hello.py' comment, a 'print' statement, a 'for' loop, and an example of string and integer concatenation and addition. The status bar at the bottom right shows 'Ln: 1 Col: 0'.

```
1 # hello.py
2
3 print('hello world')
4 for i in range(5):
5     print(i)
6
7
8 # example of different types
9 x = '42'
10 y = 13
11
12
13 # need to convert to the same type
14 print(x+str(y)) # concat as strings
15 print(int(x)+y) # add as numbers
```

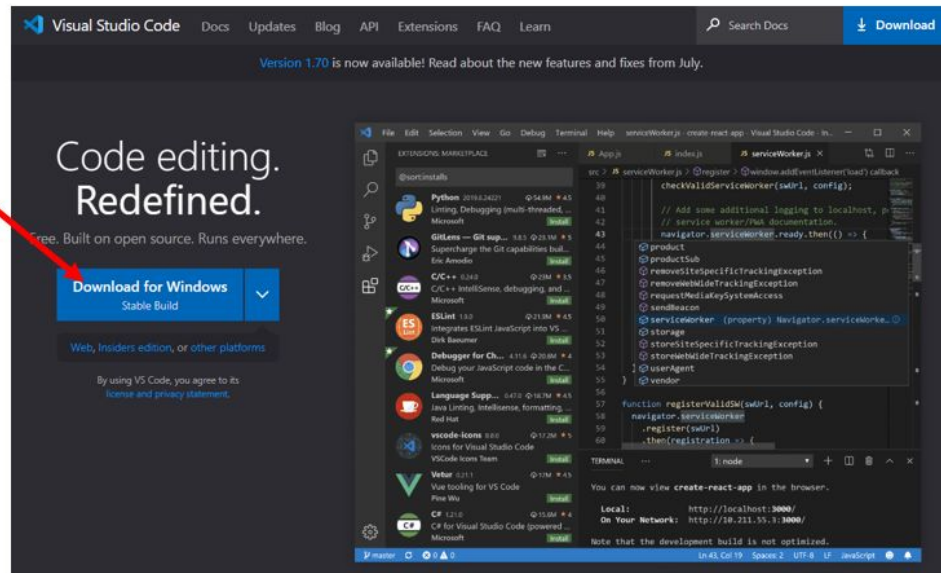
Whilst this is perfectly fine, the IDLE is rather basic and the market has developed a number of editors with significantly more features.

At the time of writing, the most widely used editor is visual studio code, which is more commonly referred to as VS code. VS code is a source-code editor made by Microsoft for Windows, Linux and macOS. In particular the features include support for debugging, syntax highlighting, intelligent code completion and snippets among others. Plus it comes with a plethora of additional functionality by way of extensions.

Some surveys suggest that VS code is used by 70% of the market, so those features have proved to be popular. We shall therefore use this editor, but note that there are many other options.

Again, the download is relatively straightforward by “googling” VS code and navigating to the website and hitting the download button.

Click here to download and install.



And we are done. So, although we could write a book or endless web pages on editors, what we have is sufficient to continue for the purpose of a coding journey with python.

# Python Basics

With Python installed and a suitable text editor in place we are now ready to code. And the first place to start will be to get a `hello world` program working.

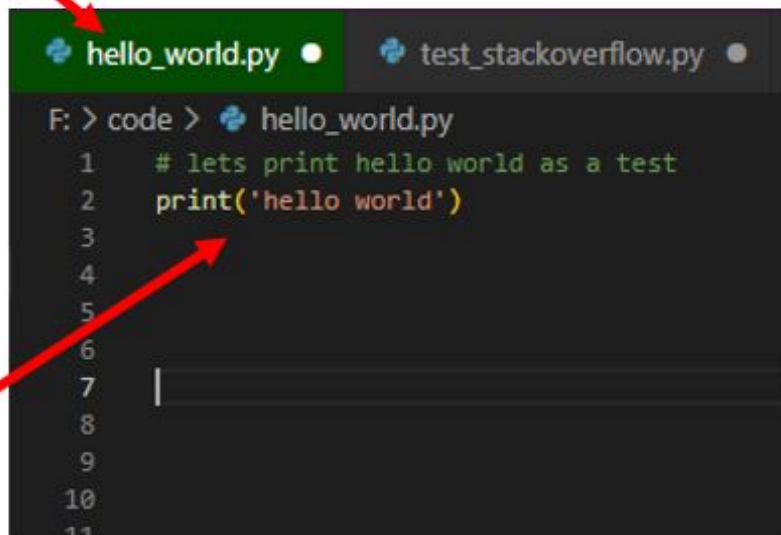
This is a simple way of testing that the **environment** works.

## Hello world

The beauty of python, is that this can be done in one line of code. We create a file and give it a name, most suitably `hello_world.py` and off we go.

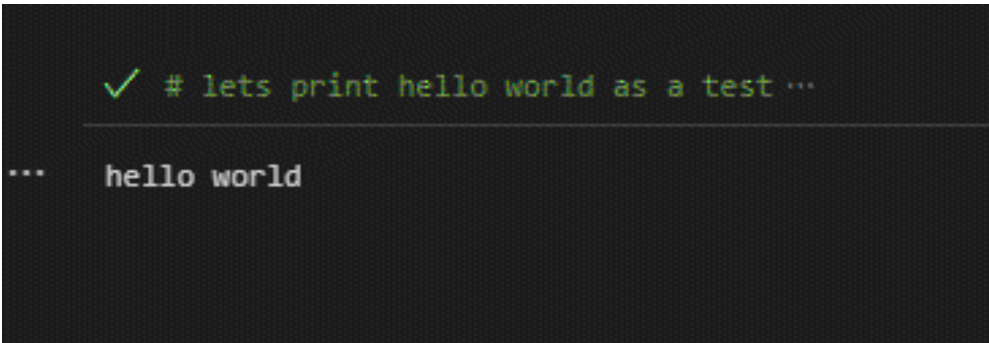
The file we  
created

The code



```
F: > code > hello_world.py
1  # lets print hello world as a test
2  print('hello world')
3
4
5
6
7  |
8
9
10
11
```

The user can then run the code and it will print to the terminal (or interactive console) hello world.



```
✓ # lets print hello world as a test ...  
... hello world
```

The various operating systems, downloads, editors, colour schemes and so forth are dependent on each individual user's resources and preference, but whatever that is, we want to get to this point.

**Getting to this point is key to everything as we are now able to code!**

## What is available

There is tons of support material and videos on the web and the reader is advised to use as many of those as necessary, nevertheless any good python book should run through the basic building blocks of the language.

Believe it or not, most languages, for the most part, are relatively simple to pick up in the early stages. A bit like a foreign language where there are some core words that can get a tourist by. For example “how much is this bread?”, if you know the words for “how much” and “bread” then you have sufficient tools to ask the shopkeeper for a loaf of bread.

Also, because languages are related, it is also the case that we can share some words and structure from other languages.

Again the same is true for programming. For example, my journey was from Basic to Fortran to C to C++ then to Visual basic, then to JavaScript, then to NodeJs and finally on to Python and along that way also touching dozens of other languages (like C# and PHP). And although each of those languages looked different, the core principles were always the same and the similarity was always over 50%.

So with this comfort, let's dive in.

## Comments

Comments are for humans to read, they do not interfere with the code in any way. Generally they are used so that we can remember or give hints to what we have done especially when we come back to the code six months later or alternatively when somebody else reads the code and wants to know what is going on or if what we have done is complicated.

There are two types of comments in python. **Single line** and **multiline** (block comments). In python they look like this.

The single line comment starts with a #.

```
# this is a single line comment
```

The Multiline (block) comment starts with three quotation marks and ends that way too. So six quotation marks in total.

```
"""  
this is also a block comment  
it uses quotation marks "  
and is a bit more descriptive  
"""
```

Or it can start with three single quotes.

```
"""  
this is a block comment  
it uses single quotes '  
and is a bit more descriptive  
"""
```

The multiline comments are often used as doc strings for functions. These are basically blocks of text explaining how the function works and are very helpful for other users of the function.

## The data types

There are 5 main types, string, integer, float, bool and None.

1. String: A string is a character or series of characters.
2. Integer: Any positive or negative whole number
3. Float: A number with a decimal
4. Bool: True or False, but can also be 1 or 0
5. None: Used to define a null value, or no value at all.

```
s = 'hello world' # this is a string
i = 100          # this is an integer
f = 10.5         # this is a float
b = True         # starts with a capital
n = None         # starts with a capital
```

## Variables

This takes us nicely onto variables. You will notice in the example above that each datatype was allocated a letter.

I had just chosen those letters because they had meaning in the context (i.e. “s” for string, “I” for integer and so forth).

But I could have just as well done this:

```
my_string = 'hello world' # this is a string
an_int = 100              # this is an integer
a_float = 10.5            # this is a float
boo = True                # starts with a capital
nothingHere = None        # starts with a capital
```

You can think of a variable as a bucket that can be filled with something. In python we say that a value is assigned to the variable. So `an_int` is assigned the value `100` and `boo` is assigned the value `True`.

Notice that although the “equals to sign” is used, I was careful not to say equals to. There is a subtle difference and this is how python works.

The “buckets” that were mentioned above are actual parts of the memory space, so literally physical memory addresses in the processor. And the values get assigned to those memory spaces which have the given variable names.

This is a very simplistic overview, so the reader is encouraged to research more, but a good starting point and sufficient for what we will do.

One of the reasons that python is a bit easier than other languages to pick up is that the coder does not have to declare a type. In fact, the type is inferred from the value given and this means that the **type of any variable can change**.

```
# x starts as a string, but then becomes an integer
x = 'hello world'    # x is a string
x = 100              # now x is an integer
```

This has advantages in particular that it is cleaner (less cluttered) and quicker to write code and disadvantages because the coder may accidentally change a variable type that *structured language* might pick up on.

In the example above, python simply allows the user to change the “x” variable type from a string to an integer without warning the user that the type has been changed.

This could have been an accident on the part of the user or this could be convenient for the user.

In terms of naming variables, you will notice that I had used different conventions. The common ones are this:

1. camelCase
2. PascalCase
3. snake\_case

The convention in python is to use snake\_case for variables and PascalCase for classes (which we will come to later).

The coder can use numbers in variable names too as long as the numbers do not precede the name.

```
# this is allowed
hello_01 = 'hello world'
hello_02 = 'hello mars'
hello99jupiter = 'hello jupiter'

# but this is not allowed
```



```
99hello = 'hello world'
```

There are quite a few more rules for naming variables, but these are the main ones and it is left for the reader to experiment.

The one thing that the reader should take from this is that **variable names should be meaningful**. It is easy to get away with single letter variables in examples, but imagine code that is 1000 lines in length with many variable names and lots of complicated work.

Which is clearer: house\_value or hv ?

## Lists, Dictionaries and Tuples

Variables are extremely useful as they enable the user to do quite a lot already, but the power of all coding languages is increased with lists.



Notice that the term List is used and **not** array.

The reason for this is that *an array is a "list of lists" with the length of each level of list the same.*

A list is a data structure consisting of an ordered set of elements, each of which may be a number, another list, etc. A list is usually denoted (a<sub>1</sub>, a<sub>2</sub>, ..., a<sub>n</sub>) or {a<sub>1</sub>, a<sub>2</sub>, ..., a<sub>n</sub>}, and may also be interpreted as a vector.

In simplistic terms, using the bucket example, a list is like a bigger bucket that contains all the smaller buckets.

A list might look like this:

Index	0	1	2	3	4
number	10	20	30	40	50

Where each item in the list has an index (its position in the list) and a number which is the value contained at that position.

The list in python is very convenient, because each element (i.e. each bucket) can actually contain any data type.

```
list_of_ints = [1, 2, 3, 4, 5, 6]
list_of_strings = ['a', 'b', 'c', 'd', 'e', 'f']
list_of_words = ['hello', 'how', 'are', 'you']
list_of_bools = [True, True, False, True, False, True]

# the types in the list can be anything
mixed_list = [1, 'hello', True, 100.5, 'world']
```

We can even have an empty list, but just using the square brackets but adding no elements inside them.

```
new_list = []
```

Not only that, but we can now create Arrays which by the definition above are **lists of lists**.

```
my_array = [list_of_ints, list_of_strings, list_of_bools]
```

And we can get really clever by putting lists into lists into lists etc. 3-D arrays, n-D arrays.

All of a sudden our very basic data types have blossomed into powerful collections of data in a short space of time.

Moving on from the list, we have the **Dictionary**. Dictionaries in python are used to store data values in {key: value} pairs. A dictionary is a collection which is unordered, changeable and does not allow duplicate keys (i.e. the keys in a dictionary are unique).

Because the word dictionary is rather a long one which is used often, we often shorten this to **dict** which is in fact the name (or type) of the dictionary class.

If we go back to the bucket analogy, the only difference between the list and the dict is that with the dict the buckets now have specific names. So instead of the indices being the position in the list, the index is now the specific name (the key) of the bucket element.

An empty dict looks like this:

```
my_dict = {}
```

From here we can add items to the list remembering that each item is a key: value pair.

```
my_dict = {'a': 100, 'b':200, 'c':300}
```

The keys can be anything so long as they are unique. If the coder was to duplicate an item either accidentally or on purpose, then the last item in the dict would overwrite the earlier one.

```
my_fruits = {'apples': 100, 'bananas':200, 'crrts':300}  
my_numbers = {0:'hello', 1:'world', 2:'mars', 3:'mars'}
```

Notice how the keys and the values can both be any valid data type (string, integer, float, bool, none).

```
random_dict = {0:'c', 1.1:None, 2.5:'d', 3:True}
```

If we go back to the my\_numbers example above, we see that this is in fact nearly identical to the list because the keys are sorted in numeric order.

So we see that there are strong similarities between **lists** and **dicts**. And in fact we could consider a list to be a special type of dict.

```
my_list = ['hello', 'world', 'mars', 'mars']  
my_dict = {0:'hello', 1:'world', 2:'mars', 3:'mars'}
```

The reason that we can say this is because of how we call the items when we want to get their values.

To get a specific list item, the coder needs to know its position in the list (its index). And call it that way. So the first item in the list is called like this: `my_list[0]`, the second item `my_list[1]`, and so forth...

If we look at the dict, this is exactly the same: `my_dict[0]`, `my_dict[1]`, and so forth...

So we can see how an ordered dict (which starts with the first key = 0) is identical to the list.

Finally, we have the Tuple. I have left this to last, because in a way it is covered by the list. The main difference being that a Tuple looks like a list but the elements cannot change, but has all the same properties. So it is like a fixed list as we cannot change the values or add or remove elements from a Tuple in the same way that we can for a list.

```
my_tuple = ("apple", "banana", "cherry", "apple")
```

This is useful in cases where we know that we do not want to change the list size, for example in a 2D plane we only want x and y coordinates and this is best represented with the tuple (x, y) as it will never change its size.

Note, there is also the 'set', which is left as an exercise for the reader to follow up on which also has good use cases (for example sorting), but it occurs less commonly. So whilst noted we delve into sets less frequently.

In summary these are the 4 collection data types in python with their common features:

1. **List** is a collection which is **ordered** and changeable. *Allows duplicate members.*
2. **Tuple** is a collection which is **ordered** and unchangeable. *Allows duplicate members.*

3. **Set** is a collection which is **unordered**, unchangeable, and unindexed. **No duplicate members.**
4. **Dictionary** is a collection which is **unordered** and changeable. **No duplicate members.**

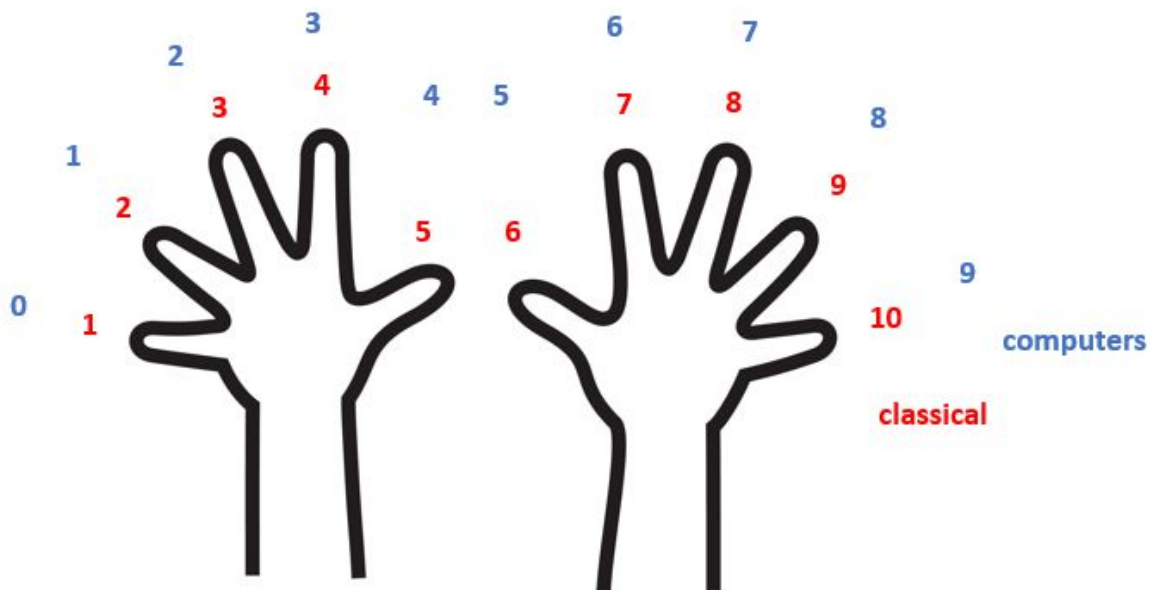
### How to count

Just a quick note on counting, as this will crop up time and time again. We as human beings have learnt to count, since we were children on our fingers. It is pretty easy, right?

1, 2, 3, 4, 5 etc ...

Well, the reality is that we have been counting the wrong way for generations. Especially given that we live in a base 10 world, we should be starting from 0 and counting to 9. The number 10 is a new line in base 10, whereas on our fingers it is the end of the first line (our pinky finger).

So by habit we often forget that lists and arrays start at the zeroth element and this leads to silly bugs and edge cases for nearly all languages.



A list of length 10 in python has 10 elements, which actually range from 0 to 9 like this: `[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]`. It is important to remember this

common pitfall and if you want to score highly on the stack, this is a good point to learn.

## Mutable and immutable

So we have been through the classical types of data: strings , integers , floating numbers , Booleans & None's . And we have taken these numbers and put them into new types as collections of data called lists , dicts , tuples and sets .

We discussed that a variable is allocated a piece of memory space and drew an analogy to a bucket. And then we discussed that those buckets could be put into bigger buckets (the collections) which are also allocated space in the memory.

So let's take a simple example.

```
# %% the memory location of a variable is mutable
x = 10      # assign x the value of 10
print(id(x)) # print the ID (memory location) of x
x = 11      # assign x the new value of 11
print(id(x)) # print the ID (memory location) of x
```

The result is this:

```
2863537193488
2863537193520
```

We see that the memory location of x had changed when its value changed from (10 to 11). So the memory location of a variable is mutable.

Now let's do the same with a list.

```
# %% the memory location of a variable is immutable

x = [1,2,3] # assign x the value of [1,2,3]
print(id(x)) # print the ID (memory location) of x
x.append(4)  # add an element to the list [1,2,3,4]
print(id(x)) # print the ID (memory location) of x
```

The result is this:

```
2864017202368
2864017202368
```

We see that even though we have changed the value of x from [1,2,3] to [1,2,3,4], the memory location of x remained the same. So the memory

location of a list is immutable.

Mutable is a fancy way of saying that the internal state of the object is changed (i.e. mutated). So, the simplest definition is: **An object whose internal state can be changed is mutable**. On the other hand, **immutable doesn't allow any change in the object once it has been created**.

At this point we just know from our observations above that memory addresses for collections like lists and dicts are immutable and therefore behave differently from variables like integers, strings and Booleans which are mutable.

But this will become an important point when working with lists or dicts as sometimes we can use this behaviour to control the efficiency of our code.

We will revisit this feature when we talk about functions which are blocks of code that take inputs (called arguments) to do a process and return something back. As it is commonplace to use variables and collection at the function arguments and the memory address (mutability) will make a difference to the behaviour of the code.

## Basic algebra

Python does all the basic operations out of the box as like any standard language. In fact the dynamic typing makes it convenient as variables do not need their types to be assigned before they are used.

It works the same as calculators have done since their invention following bodmas rules.

### Ordering Mathematical Operations

B	O	D	M	A	S
Brackets (...)	Orders $\sqrt{x}$ $x^2$	Division $\div$	Multiplication $\times$	Addition $+$	Subtraction $-$

So we can do the usual operations conveniently.

```
a = 5 + 3 - 2      # adding and subtracting
b = 3 * 5 / 2      # multiplication and division
c = (3 + 5) / (2 * 2) # brackets
d = 3**2           # powers

print(a,b,c,d)

# returns this: 6 7.5 2.0 9
```

and basic algebra

```
z = (a + b) / (c * d)
print (z)

# returns this: 0.75
```

So there is now a lot more that we can do.

## For and while loops

Some would argue that a lot of the power of computing begins here. One of the biggest gains of computers over humans is the ability to repeat tasks fast without making mistakes.

```
# %% for loops
for i in range(10):
    print(f'hello {i}')
```

This simple piece of code returns this list in a fraction of a second.

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

The reality is that we could print more words than the entire contents of this book in a fraction of a second. The computer used to type this book can



produce 100,000 hello n sentences in 0.25 seconds.

This equates to 400,000 operations per second. So we have immense computing power on a standard device at our fingertips. A decent start.

We will learn to harness this ability.

In other languages like visual basic or c++ , a for-loop has a structure that looks like this:

```
// for loop in c++  
  
for (int i = 0; i < 5; i++) {  
    // the block of code goes here.  
    cout << i << "\n";  
}
```

Where we have a start , stop and increment in the parenthesis and the block of code in the curly braces.

In python, the syntax is cleaner. There are no curly braces which are replaced by the indentation and the loop actually iterates over an object. We have used the range() function which creates a convenient sequence of numbers range(start, stop, step) that does all the same things, but in fewer lines of code, which is convenient and easier to read.

In fact, we will see that python loops are even more convenient in this manner as we can iterate over any collection such as lists and dicts .

We do it quickly like this:

```
# lets make a list  
my_list = ['a', 'b', 'c', 'd']  
  
# now iterate over the list  
for x in my_list:  
    print('hello', x)
```

Which returns this:

```
hello a  
hello b  
hello c  
hello d
```

The other powerful type of loop is the `while` loop. We can say that the `for` loop is **bounded** as it starts at a certain point and goes up to a certain number and then terminates when it has reached the end. So, we saw that `for i in range(10)` did 10 iterations. It got to the end. If we had set 100 or 1,000,000 it would have done that many loops and then stopped.

On the other hand the `while` loop keeps on going, so it is **unbounded**, until a condition is met.

```
x = 0
while x < 10:
    print('hello ' + str(x))
    x = x + 1
```

returns this:

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

We have to be a bit more careful with `while` loops because of their unbounded nature. For example, in the example above, if the line `x = x + 1` was removed or set to `x = x - 1`, then the condition that `x < 10` would persist forever and the loop would never finish.

The program would hang.

Quite often we do something like this:

```
i = 0
x = 0

condition = True
while condition:
    x = x + 10
    i = i + 1
```

```
if x > 100:
    # the condition changes
    condition = False

print(i, 'loops done')
```

The while loop runs when the condition is true , then at some point it becomes false and the loop is exited. Again, we need to be careful that the condition can be exited otherwise the program will remain in the loop forever (hanging).

## If else logic

The “ if” statement comes up everywhere in coding. The format and logic is rather straightforward:

```
if x > 10:
    print('x is big')
```

it can be combined with an else statement.

```
if x > 10:
    print('x is big')
else:
    print('x is small')
```

Again, in python the indent indicates the block of code where other languages would use curly braces.

We can combine if statements or use the elif keyword.

```
if x > 10:
    print('x is big')
elif x < 2:
    print('x is small')
else:
    print('x is medium')
```

Or this

```
if x > 10:
```

```
    print('x is big')
if x < 2:
    print('x is small')
else:
    print('x is medium')
```

We can also nest if statements

```
if x < 10:
    if x > 2:
        print('x is medium')
```

the nested if behaves like an AND logic operator because two criteria need to be satisfied for the print statement to be invoked.

## Functions

We now come to one of the most important features of all programming. A function is a **block of code** which only runs when it is called. You can pass data, known as parameters, into a function. A function can return data as a result.

Using this we can break the code up into workable parts, like little modules and then call those modules as we need.

The function has the same indented block, so looks like this:

```
def my_function():
    print('hello world')
```

We run the function at any time in the code by calling it.

```
# define the function
def my_function():
    print('hello world')

# call the function
my_function()
```

We can call the function as many times as we like as it is a reproducible block of code:

```
# define the function
def my_function():
```

```
print('hello world')

# call the function 10 times
for i in range(10):
    my_function()
```

returns this:

```
hello world
hello world
hello world
hello world
hello world
hello world
hello world
hello world
hello world
hello world
hello world
```

Just like the ‘ for ’ and ‘ while’ loop and the ‘ if ’ statement, once the block (defined by the indent) is completed, the function is over.

The function can have inputs called arguments:

```
name = 'world'          # a variable

def my_function(name):  # function takes variable
    print('hello ' + name) # function uses variable
```

The function is often commented with a string, known as the docstring which describes to the user what the function does:

```
def my_function(name):  # function takes variable
    """ prints hello & the name """
    print('hello ' + name) # function uses variable
```

The function can return something back.

```
name = 'world'          # a variable

def my_function(name):  # function takes variable
    """ takes a name, returns a string
```

```

    inputs: name (str)
    returns: s (str)
"""

s = 'hello ' + name    # concatenates a string
return s               # function returns a string

```

We can get assign the return value back to a variable like this:

```

def add_numbers(a, b):
    """ add a an b, returns the answer """
    return a + b

my_sum = add_numbers(1, 3)    # value returned

print('the sum is', my_sum)  # print answer

```

Returns this:

```

the sum is 4

```

Functions can call other functions that have been previously defined, so we can control the flow of the code nicely in modular blocks.

```

def add_numbers(a, b):
    """ add a an b, returns the answer """
    return a + b

```

```

def my_name(name):
    """ takes a name, returns a string

    inputs: name (str)
    returns: s (str)
    """

    s = 'hello ' + name
    return s

```

```

def main():
    """ calls both the above functions """

```

```
z = add_numbers(7, 8)

if z > 10:
    s = my_name('world')
    print(s)

# call the main function
main()
```

A collection of lots of useful functions can be considered to be a *library* (or a *module*). We could save this module of really useful functions and call it later (i.e. we can `import` the module) in another piece of code.

## Classes

Hot on the heels of function are classes. Classes are the foundation of many object oriented languages. One might consider a class as a *user-defined blueprint or prototype from which objects are created*. We will see what this means.

We will look at an example, but note that there are many examples. Often, the examples that we see are about people, cars, or animals (a dog is a good one). In fact anything that has `attributes` and `methods` that can be replicated will perfectly fit into the class system which is why it is so powerful.

Before we do this, let's see what a class looks like in python. The code block looks like this.

```
class MyClass:
    """ this is a class """
    pass
```

We can add attributes and methods (the functions).

```
class MyClass:
    """ this is a class """
    pass
```

```

x = 10    # variable is called an attribute
y = 100

def add_one(self):
    """
    this function is a 'method' of the class
    x is incremented by 1
    """
    self.x = self.x + 1
    return self.x

```

Here we have added two attributes `x` and `y` and one method `add_one()` which adds 1 to the number that is entered. We also note the convention (but not a rule) is to use a capital letter to name a class. And we see a new keyword, `self`.

The `self` parameter is an important feature of the class as it refers to itself and other Object Oriented Programming (OOP) languages use similar terms (for example `nodejs` uses the `this` keyword to refer to itself).

We can create an instance of the class by calling it. This is called instantiating the class.

```

x = MyClass()

```

and we can call its attributes and methods easily.

```

class MyClass:
    """ this is a class """
    pass

x = 10    # variable is called an attribute
y = 100   # here is another attribute

```



```

def add_one(self):
    """
    this function is a 'method' of the class
    x is incremented by 1
    """
    self.x = self.x + 1
    return self.x

# create instance of the class
something = MyClass() # instantiate class

print(something.x)    # this prints 10
print(something.y)    # this prints 100

something.add_one()   # execute the method (adds 1)
print(something.x)    # this prints 11

```

In a nutshell, the above represents an overview of the majority of what a class is and how it can be used. Again, for the purpose of the book we stick just to this description, but noting that one could in fact write an entire book of classes alone. And it is for the reader to follow up on this.

## Magic methods

When we look at classes we will notice that some functions begin with a double underscore `__`. They are also known as dunder methods which is a mashup of double and under.

They can help override functionality for built-in functions for custom classes. Common methods are:

- `__str__` : for strings
- `__len__` : number of items
- `__dict__` : dictionary of the item

For classes, we often see, right at the top `__init__()` : which indicates that Python will use the method internally. The user should not explicitly call this method. And since this method is immediately called after creating a new class, we can use this to initialise the object's attributes.

```
class Node:
    """ node for the tree """

    def __init__(self, data):
        """ construct class """
        self.left = None
        self.right = None
        self.data = data

    def PrintTree(self):
        """ another function """
        print(self.data)
```

We will not dig deeper into this at this point for two reasons.

1. There is too much to write about in one book.
2. We will certainly run into them later.

So at this point, it is just a heads up about a feature that we should be aware of.

## Importing modules

At the top of nearly every piece of code you will tend to see a bunch of import commands. It is convention to place all import statements at the beginning of a module or script.

One reason that we do this is because it helps us (the users or the next programmer) to remember what we imported externally from the code. These are external files, called `dependencies` as the code will not run without them that is being imported into the code's ecosystem.

Common modules are: `time` , `datetime` , `math` , `random` , `statistics` , `requests` , `pandas` , `numpy`...

```
import time    # a module of useful time functions

t = time.time()
print(t)

# the result
# 1660605141.3659403
```

In fact the list is nearly endless as there are over 200 standard libraries and over 137,000 external libraries. This is one of the major attractions of a language like python which has a massive community with immense support.

A standard library is one which comes already packaged with python so is already on the desktop and available to use when python is installed and there are fewer of these otherwise the download would be very data consuming.

The external libraries are relatively easy to install with something called a package manager ( pip or conda ), which is usually done with one line in a command prompt.

The modules usually have relevant names that give a clue as to what they do, but also have the same descriptions and docstrings so that the user can verify if they are suitable for the task at hand.

Examples of internal modules:

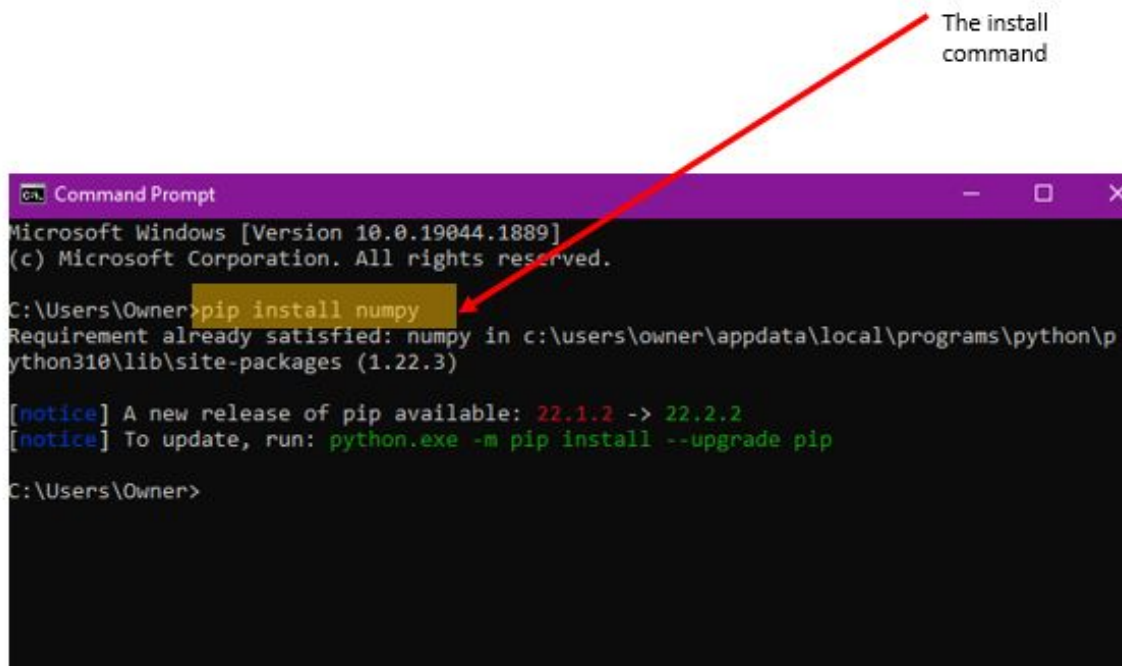
- **Time**: provides various time-related functions
- **Datetime**: supplies classes for manipulating dates and times
- **Math**: access to the mathematical functions
- **Random**: implements pseudo-random number generators for various distributions
- **Statistics**: provides functions for calculating mathematical statistics

Examples of external modules

- **Requests**: a simple, yet elegant, HTTP library.
- **Pandas**<sup>[1]</sup>: powerful Python data analysis toolkit

- **Numpy**: the fundamental package for array computing with Python
- **Scipy**: Scientific Library for Python

Two of the most common external libraries are numpy and pandas and here is an example of the install in windows.



```
Command Prompt
Microsoft Windows [Version 10.0.19044.1889]
(c) Microsoft Corporation. All rights reserved.

C:\Users\Owner>pip install numpy
Requirement already satisfied: numpy in c:\users\owner\appdata\local\programs\python\python310\lib\site-packages (1.22.3)

[notice] A new release of pip available: 22.1.2 -> 22.2.2
[notice] To update, run: python.exe -m pip install --upgrade pip

C:\Users\Owner>
```

In fact pandas and numpy are so commonly used that they come installed with the Anaconda package which describes itself as “The world’s most popular open-source Python distribution platform”.

Basically in simple terms, we like to keep, use and repeat the good code so that we don’t continuously waste our time reinventing the wheel.

When the code gets sufficient and complete enough, it becomes worth packaging this into a library.

Python along with all other modular languages benefit greatly from these libraries as the user only needs to install the core components that are relevant to them.

For example, **let's generate a list of random numbers and work out the min , max , avg & std of that list and then see how long it takes to perform this operation.**

```
# the imports
import time      # module for time
import random    # module for random numbers
import statistics # statistics module

t0 = time.time()      # the start time

rand_list = []        # an empty list
for i in range(100_000): # loop 1000 times
    r = random.randint(0,100) # a random integer
    rand_list.append(r)      # add this to the list

# print the results
print('min:', min(rand_list))
print('max:', max(rand_list))
print('avg:', statistics.mean(rand_list))
print('std:', statistics.stdev(rand_list))

t1 = time.time()      # the end time
time_taken = round(t1-t0, 2) # time taken (2dp)

print('the time taken is', round(t1-t0, 2), 'seconds')
```

The results look like this:

```
min: 0
max: 100
avg: 50.10659
std: 29.136766431494777
the time taken is 0.24 seconds
```

So we have generated a random list of 100,000 integers and computed basic statistics on this random list in less than  $\frac{1}{4}$  of a second. And all of this is done in less than 20 lines of code.

In fact, with list comprehensions and a bit of tidying up, this could be done in 10 lines of code. And if we deployed the numpy library, the speed would have been faster too.

```
# list comprehension replaces for loop
rand_list = [random.randint(0,100) for x in range(1_000_000)]
```

Sometimes the library module names can be rather long, for example `random.randint()` is quite lengthy. Or `statistics.stdev()`. When repeated many times in the same line, the code gets verbose and cluttered.

Fortunately python offers an alias system so that we can shorthand the longer names to more convenient names.

```
import random as rd    # alias is assigned: rd
import statistics as sts # alias is assigned: sts

r = rd.randint(0,100) # we can use the alias rd
```

A very popular library is `matplotlib` which is used for plotting charts. The typical module import looks like this:

```
from matplotlib import pyplot as plt
```

Which looks like a handful for one line of code, but is in fact quite simple. Because `pyplot` is simply a function that lives inside the `matplotlib` module and `plt` is just the alias we assign to the thing that we want. We shall look into this more later on.

Finally, a common pitfall on importing modules. Python allows for wildcard imports. This means that the user can do this:

```
from math import *
from os import *
from tkinter import *
```

The wildcard ``*`` means that all the methods will be imported. So for example, instead of doing this:

```
x = math.cos(90)
```

We can now do this:

```
y = cos(90)
```

There is clearly a level of convenience that comes with this as the former is more longwinded than the latter. And actually, for smaller modules this would be sufficient.

However, the drawback is that without knowing the myriad of methods that are contained in the module, we could easily overwrite them and furthermore, it is no longer clear where `cos()` came from. Again, for small pieces of code, we could say that this is easy as `cos()` is clearly a maths function and has come from the maths module, but look at this error.

```
from PIL import Image, ImageTk
from tkinter import *
```

The `tkinter` module contains a method called `Image`, but the coder thinks that the `Image` method has been called from the `PIL` module<sup>[2]</sup>.

Because Python allows for reassignment the first import of the `Image` function is overwritten by the second one without warning and this was probably unintended.

So it is helpful to be clear with imports and in this case an alias would have been more suitable and removed the ambiguity:

```
import tkinter as tk
```

And this is clear.

## Logical order

We learned earlier that python code is executed line by line in sequential order.

This means that the order matters. For example we can not use a variable before it is defined and similarly we cannot use a function before it is defined.

This takes some getting used to compared to other languages where the functions are hoisted which means that with other languages, we can dump the big bulky functions at the bottom of the code. In python we can not do this as the need to be read first.

for variables:

```
x = 10    # define x
print(x)  # this works, x was defined
```

```
print(y)  # this fails, y is not defined yet
y = 10    # define y, but too late !!
```

We see that this throws a `NameError`, which means there is an error with the name. It was not defined at the time it was run.

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

and for functions:

```
# the function has not been defined
# so it fails
print( add_numbers(1, 2) )

def add_numbers(a,b):
    return a + b

# we can call the function here
print( add_numbers(1, 2) )
```

Because of this, python behaves in a `synchronous` way. what comes first goes first. So the user must be careful of order and can not be casual.

This can sometimes be frustrating as often it can be helpful to put the main body of code at the top and call the functions later (especially for those who have come from languages like visual basic and node.js).

However, there is a neat little trick in python which is to wrap the invocation into a function on its own, and then the user can call that function at the end.

It looks like this:

```
import math

def main():
```



```

"""
    This is the main body of the code
    where the control flow is.
    we like to see the main control at the top.
"""

print('...lets do many things')

# now call the function
x = add_three_numbers(10,20,30)
print(x)
print('...do more things')

# and call another function
y = circle_features(x)
print(y)

# the functions that we use later go here

def add_three_numbers(a,b,c):
    return a + b + c

def circle_features(radius):
    area = math.pi * radius**2
    volume = 4 / 3 * math.pi * radius**3

    return area, volume

# we call the main function here.
main()

```

## Making modules

So we can import modules that we made by the community, for which there are thousands. These modules were very powerful collections of useful functions with use case examples and plenty of support.

However, the modules created only serve to assist the user in building their own project. The project could be one snippet of code in one file of no more than 20 lines of code or it could be a large meaningful piece of code spanning many thousands of lines and also have user created modules of its own.

Python caters well for both of these and in fact makes it very easy for the user to both create and import their own modules.

Every file that a user creates is saved as a \*.py file and this will live in a folder. So for example, the “hello\_world.py” would have been saved somewhere on the pc.

Let's assume that it is saved on the C drive (for a windows based computer) in a folder that we named “code”. Then the full path would look like this:

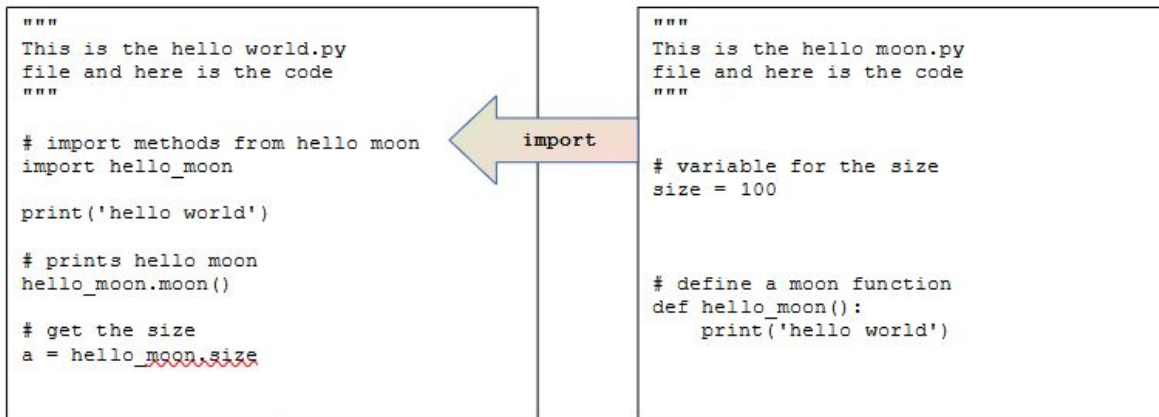
C:\code\hello\_world.py

We can say that the python file is called hello\_world.py and the project folder is c:\code . Now, if we create another file in the same folder, say hello\_moon.py then we would have two files in the project folder.

And all we have to do is this:

hello\_world.py

hello\_moon.py



So we can see that the import statement brings in all the attributes and methods from one file ( hello\_moon.py ) into the other file ( hello\_world.py ) in just one line of code.

The rules are very simple too. The imported folder should be in the same folder or a sub folder of the project folder. That is it.

So we now have the tools necessary to create an entire project which is broken into useful and meaningful modules.

## The main module

There is one final part to all of this, which is that we now have a collection of attributes and functions and collections of modules (the python files) in various folders (hopefully not too many otherwise the project is complicated). So how do we determine the starting point in all of this?

There is usually one file which is deemed to be the main file or we could say the entry point to the code. the file could have any useful or descriptive name that the user wants.

Having read the **magic methods** chapter we are aware that all objects in python have special built in functions which are denoted with the double

underscore. Well, the file object is no different. Here are some of the more useful ones:

- `__file__`: the path name of the file.
- `__doc__`: a description of the file.
- `__name__`: the name of the file

We could actually get a list of these by typing `print(dir())` in a file and running the code.

The interesting one here is the `__name__` attribute as this tells us the name of the file that was imported. It gets its value depending on how we execute the containing script. So, the actual **main file** (which is not imported) is given the name `__main__`.

We often see this at the bottom of the main file in nearly every large project.

```
if __name__ == "__main__":  
    # run some code  
    print('do something')
```

The code inside the `if` statement is now only run when the filename is that of the main file. Our entry point !

As a side note, we could import files from other folders that are not contained within the project folder by adding an additional path name like this.

```
# some_file.py  
import sys  
# caution: path[0] is reserved for script path (or " in REPL)  
sys.path.insert(1, '/path/to/application/app/folder')  
  
import file
```

But for the purpose of what we do and *best practice we like to keep the package together* in one folder.

## Coding Techniques

So at this point we have had an overview of python, how to set up the environment and also an overview of the basics of language itself.

We looked at variables, collections, functions, loops and if statements among others. Everything that we have seen has been relatively top level as a deep dive into even just one topic could be a book in itself.

It would be wise for the reader to recap and support their book reading with actual real world testing on a device. Yes, this means **writing code yourself**, starting with the `hello_world.py` program. This cannot be stressed enough, you now have to get your fingers onto a keyboard to enrich the benefit of this read.

Having said this, it is now time to look at some problems that we tend to encounter. I am not sure how I will break these down into categories.

The important thing to remember is that these “problems” should be considered to be fun brain teasers and other interesting cases.

## What is a coding problem

It is any kind of problem and usually interesting ones that we might not be able to solve with a pen and paper alone. Initially, when trying to learn, I took the 100 daily coding questions. That basically, means, every day for 100 days, pick a problem and try to solve that problem in any language.

Initially, I was using `node.js` and a friend was using `python`, so it was interesting to compare methods. The reality is that both languages (sometimes referred to as scripting languages) have a >75% similarity.

In fact, if one swaps curly braces for indents and removes the semicolons at the end of lines, then the languages are even closer.

## What is a coding technique

We could consider a technique to be a general form as to how we approach a problem. We tend to use the same techniques quite often to solve a whole

variety of problems in the same way that a particular tool in a toolbox might be used to fix many issues.

We have common techniques, like for loops and while loops just in the same way that we have screwdrivers and spanners in our standard home toolbox.

And then we have more specialised tools, such as the pandas dataframe which might be more akin to a sander on the toolbench. These do fewer things, but much better.

And sometimes we can even use the wrong tool which gets the job done, like using a screwdriver or knife to open a tin of paint.

Each will have its own merits and downfalls. However, if we can more often than not pick the right tools, then our code becomes increasingly efficient. This efficiency is especially important and noticeable when the data or the number of processes become large.

It could be something as simple as taking static variables outside of the loop. Have a look at these two snippets of code:

Variables inside the for loop

```
import time

t0 = time.time()
for i in range(1_000_000):
    a = 100
    b = 2
    if i%1000 == 0:
        print(a + b + i, end = "\r")

t1 = time.time()

print('the time taken was', (t1-t0), 'seconds.')
```

Variables outside of the for loop

```

import time

t0 = time.time()
a = 100
b = 2
for i in range(1_000_000):
    if i%1000==0:
        print(a + b + i, end = "\r")

t1 = time.time()

print('the time taken was', (t1-t0), 'seconds.')

```

Which one is quicker and why ?

If we look at the first piece of code, the variables a and b are assigned in every loop. There were 1 million loops, so 1 million additional passes of this line. Whereas, in the second piece of code, the variables have been taken out of the loop saving a massive cost in terms of programming.

We are not punished too badly in this case, because we have a small snippet of code which, in both cases, runs in less than a second. But if the code was more intensive or the variables happened to be large functions, then the code would face efficiency problems.

Here this difference was 90 milliseconds for the efficient method versus 180 milliseconds for the inefficient method.

We could make the easily highlight how bad this can be with the following:

```

import time

n = 3
def n_second_process(n):
    """ a process that lasts for n seconds """
    time.sleep(n)
    return 'hello'

```

```
x = n_second_process(n)

for i in range(100):
    print(x + ' world')
```

again this:

```
import time

def n_second_process(n):
    """ a process that lasts for n seconds """
    time.sleep(n)
    return 'hello'

for i in range(100):
    n = 3
    x = n_second_process(n) # this is expensive !!!
    print(x + ' world')
```

Here the **first piece of code takes just over 3 seconds** whereas the **second snippet takes 5 minutes** ( $100 * 3$  seconds) because the time consuming function was placed inside the loop and it could have been much worse with just a change in the loop number.

These are the easy wins that the coder needs to be aware of.

## Recursive functions

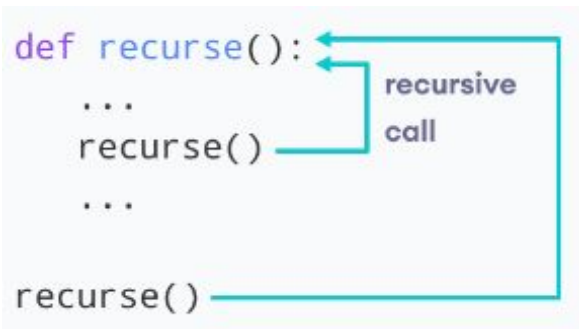
Recursion is a common mathematical and programming concept. It means that a function calls itself. This has the benefit of meaning that you can loop through data and apply logic to reach a result.

This is especially useful when the same pattern repeats itself, although an unfinished or indefinite recursion could cause a stack overflow. So the



coder needs to make sure that at some point an end condition is met.

The recursion process looks like this in python:



A popular mathematical problem solved by recursion is finding the factorial of a number,  $n$ .

Let's take a look at how the recursion process works in practice:

```
def recur_factorial(n):  
    """ calculate factorial of n """  
    if n == 1:  
        return n  
    else:  
        return n*recur_factorial(n-1)  
  
recur_factorial(1000)
```

We notice how the recursion function simply keeps returning itself, each time lowering the value of  $n$  until the condition that  $n==1$  is reached and the recursion stops returning the final value.

In this example we see that  $1000!$  is computed in a fraction of a second yielding a monstrous number that had to be shrunk in font size to fit on this page.

```
40238726007709377354370243392300398571937486421071463254379991042993851239862902059204420848696940480047  
99886101971960586316668729948085589013238296699445909974245040870737599188236277271887325197795059509952  
76120874975462497043601418278094646496291056393887437886487337119181045825783647849977012476632889835955  
73543251318532395846307555740911426241747434934755342864657661166779739666882029120737914385371958824980
```

[illegible]

Having praised the recursive function, we can see that the same can be achieved in a linear fashion with the for loop.

```
def ret_factorial(n):
    f = 1
    for i in range(1,n+1):
        f = f * i
    return f
```

We note that recursions are elegant to look at and have the advantage that they can break complex tasks into small simple units.

However, the downside of recursion is that they are expensive in terms of memory and time. In the above example, the factorial problem works well,

but a value of  $n > 2000$  would have caused a memory issue because each new function was assigned a new place in memory.

## Backtracking

Backtracking is recursive in nature. Backtracking is simply reverting back to the previous step or solution as soon as we determine that our current solution cannot be continued into a complete one.

```
# backtracking

def permute(list, s):
    if list == 1:
        return s
    else:
        return [
            y + x
            for y in permute(1, s)
            for x in permute(list - 1, s)
        ]
print(permute(1, ["a","b","c"]))
print(permute(2, ["a","b","c"]))
```

the above code returns this:

```
['a', 'b', 'c']
['aa', 'ab', 'ac', 'ba', 'bb', 'bc', 'ca', 'cb', 'cc']
```

## list comprehension

If we take a look at the backtracking algo, there is something that we have not seen before. It is a complicated version of a list comprehension .

The ‘ list comprehension ’ offers a *shorter syntax* when you want to create a new list based on the values of an existing list and is a convenient method in python which does not exist in many other languages.

Let’s do this by example: imagine that we have a list of numbers and we want to square just the even numbers. This was the old method

```
my_list = [1,2,3,4,5,6,7,8,9,10]
```

```
squared_list = []    # create new list
for i in my_list:    # iterate thru list
    if i%2==0:        # check even numbers
        squared_list.append(i**2) # append list
print(squared_list)  # [4, 16, 36, 64, 100]
```

and this is the list comprehension of the same:

```
my_list = [1,2,3,4,5,6,7,8,9,10]

squared_list = [x**2 for x in my_list if x%2==0]
print(squared_list)  # [4, 16, 36, 64, 100]
```

or lets take all of the vowels out of the words “hello world” and put them into a new list.

```
s = 'hello world'

v = [x for x in s if x in 'aeiou']
print(v)  # ['e', 'o', 'o']
```

The list comprehension would on other collections like dicts too. Here is an example of extracting farmyard animals from a dictionary where the key is the animal name and the value is how many there are.

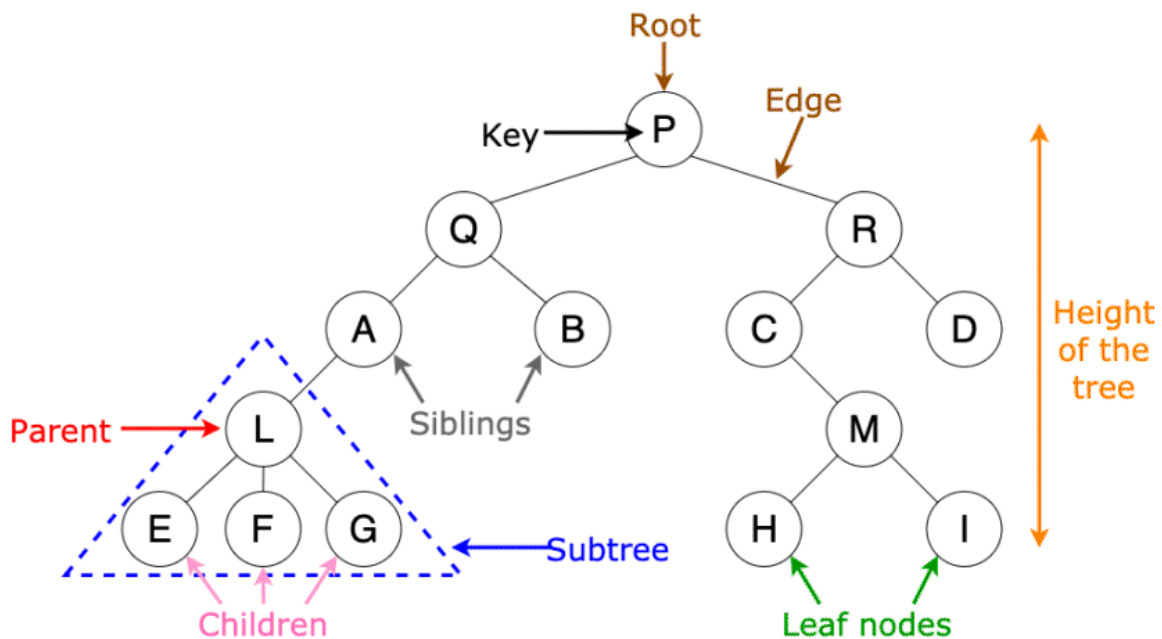
```
d = {'cats':2, 'dogs':5, 'pigs':7, 'sheep':30, 'cows':12}
e = {k:v for k,v in d.items() if v>10}
print(e)  # {'sheep': 30, 'cows': 12}
```

The reason that this is useful is because there a just so many things that go into collections

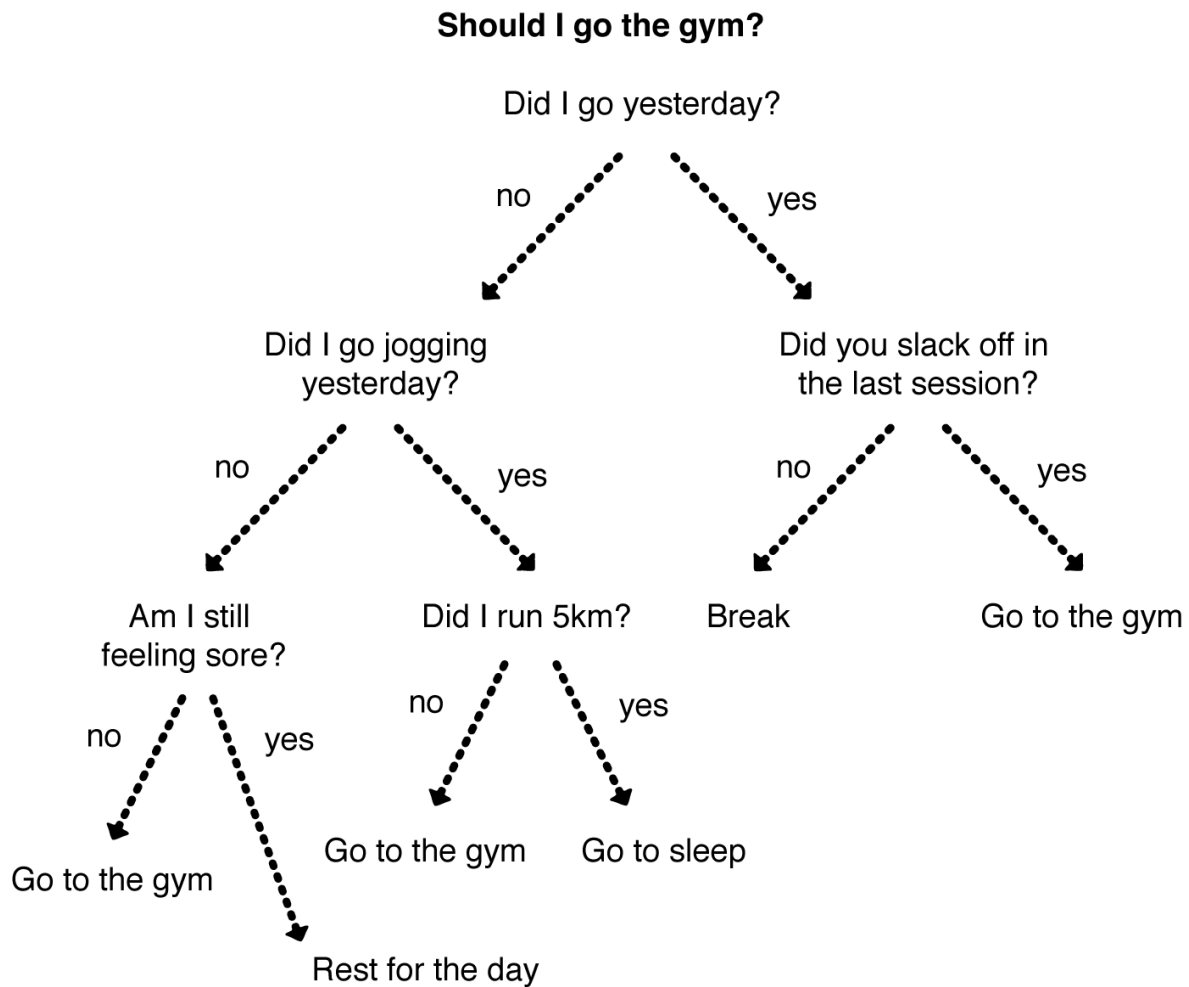
## Binary Tree

Binary trees are common features of programming as they help make decisions. The process is to start at the top of the tree, which is often referred to as the root node or the parent node , and then traverse down the tree child nodes until we reach the end nodes or leaf nodes .

Here is a picture with some of the key terms.



This mathematical tree diagram translates into a real world process like this.



Because each decision eliminates half of the remaining tree, the binary search is powerful at arriving at a solution. Because of the repetitive nature of the tree structure, one can use a recursive process to traverse through the tree nodes or alternatively an iterative search (which we have seen to be more efficient).

The search runs until we reach some leaf node at which point the final result is reached.

The binary tree is even more powerful than this as one can traverse **down the tree** and then, with a process called **backward induction**, reverse all the way back **up the tree** to the parent node. This is one possible technique for financial option pricing.

We will now use some of the techniques from the earlier sections to deploy a binary tree.

In particular, classes. the class structure is ideal for the repetitive nature of the tree as we only need one class to be the blueprint for all nodes and in python the construction looks like this:

```
class Node:
    """ node for the tree """

    def __init__(self, data):
        """ construct class """
        self.left = None # a left child
        self.right = None # a right child
        self.data = data # data at the note

    def PrintTree(self):
        print(self.data)

# instantiate node
root = Node(10) # create instance
root.PrintTree() # returns 10
```

Now that we have a blueprint for a node, we can insert this into the tree by creating an insert method in the class. And it should also be pointed out that we could have set nodes with more than two children (for example we could have included self.middle ).

```
class Node:
    def __init__(self, data):
        self.left = None
        self.right = None
        self.data = data

    def insert(self, data):
        """
        a function to Compare the new value with
```

the parent node and add this to the tree.

"""

```
if self.data:
```

```
    # whatever logic we need
```

```
    if data < self.data:
```

```
        if self.left is None:
```

```
            self.left = Node(data)
```

```
        else:
```

```
            self.left.insert(data)
```

```
    elif data > self.data:
```

```
        if self.right is None:
```

```
            self.right = Node(data)
```

```
        else:
```

```
            self.right.insert(data)
```

```
    else:
```

```
        self.data = data
```

```
# Print the tree
```

```
def PrintTree(self):
```

```
    if self.left:
```

```
        self.left.PrintTree()
```

```
    print( self.data),
```

```
    if self.right:
```

```
        self.right.PrintTree()
```

```
root = Node(12)  # create instance
```

```
# insert some nodes
```

```
root.insert(6)
```

```
root.insert(14)
```

```
root.insert(3)
```

```
root.PrintTree()  # returns 3, 6, 12, 14
```



With the tree now created we can traverse through the tree. There are three different ways that we can do this.

- **In-order Traversal:** the left subtree is visited first, then the root and later the right subtree.
- **Pre-order Traversal:** the root node is visited first, then the left subtree and finally the right subtree
- **Post-order Traversal:** we traverse the left subtree, then the right subtree and finally the root node.

This is the additional method that we would add for a pre-order traversal.

```
def PreorderTraversal(self, root):  
    res = [] # empty list to store values  
    if root:  
        res.append(root.data)  
        res = res + self.PreorderTraversal(root.left)  
        res = res + self.PreorderTraversal(root.right)  
    return res
```

And the result, is this:

```
print(root.PreorderTraversal(root)) # return [12, 6, 3, 14]
```

## Fun problems

This section is about some fun and interesting problems. There are a whole range of these and to add to that some of the problems have multiple methods to solve them, so we will only be able to cover a limited amount.

The solutions proposed might not even be the best, so the reader is encouraged to contact the Author if you feel that you have a better approach for a particular solution.

## Perfect numbers

Let's start with a little fun problem.

A number is considered perfect if its digits sum up to exactly 10. Given a positive integer n, return the n-th perfect number.

For example, given 1, you should return 19. Given 2, you should return 28, given 123 you should return 1234.

This is a good first problem, because we get to use a function in anger and we also get exposed to: types , if statements and for loops .

Here is one solution:

```
def ten(number):  
    """  
    function takes number and returns the  
    compliment number that makes the sum == 10  
    """  
    numberstr = str(number)  
    total = sum(int(x) for x in numberstr)  
    if total == 10:  
        return number  
    elif total < 10:  
        return int(numberstr + str(10 - total))  
    else:  
        return "Input number has digits that sum to more than 10."  
  
# lets test out answers  
for x in [3, 6, 142, 53453, 15, 363]:  
    print(x, "-->", ten(x))
```

The result looks like this:

```
3 --> 37  
6 --> 64  
142 --> 1423  
53453 --> Input number has digits that sum to more than 10.  
15 --> 154  
363 --> Input number has digits that sum to more than 10.
```

Let's take a look. We have declared a function and called it “ten” because it seems like a good name for the problem. The function takes one argument called “number”.

The first line converts the number into a string and we assign it to a variable called “numberstr” using the inbuilt `str()` function. We do this because we know that python can treat strings as lists of characters and we want to get each character separately, so it is a useful little trick here. We can add all the single component numbers, which we have converted back to integers using the `int()` function inside of the `sum()` function and what we recognise as something that looks like a list comprehension .

That was a mouthful to say, but we can see the power of python's conciseness which condenses everything into one logical line.

From this point we do some logic checking with the if statement. If the number is 10, then the function returns the number as there was nothing to do, else if the number is less than 10, we calculate the final number and return this and finally if the number was greater than 10 we return a message saying so.

The function is done and now we test this over an array of numbers to see how the output looks. It looks like what we were expecting. Job done.

Let's test the edge cases. What happens if we insert zero into the list? the output would return 10 which is two numbers that sum to 1, so this is a fail. Can you think of how we can prevent this ?

And could you propose a better or alternative solution?

## Swap apples for ideas

This is a pretty little anecdote which does not contain anything difficult, but displays the use of classes and the use of swapping variables (tuple swap):  
`a,b = b,a` And the use of: `if __name__ == "main__"` , which hoists the function and class.

*“If you swap apples then you don't gain anything.  
but if you swap ideas, then you gain ideas.”*

```
def main():
    person1 = Person(5, 3)
    person2 = Person(1, 2)

    # swap apples
    person1.apples, person2.apples = person2.apples, person1.apples

    # swap ideas
    person1.ideas = person1.ideas + person2.ideas
    person2.ideas = person1.ideas
    print(person1)
    print(person2)
```

```
class Person:
    """ description of the class """

    def __init__(self, apples, ideas):
        """
        apples (int): number of apples
        ideas (int): number of ideas
        """
        self.apples = apples
        self.ideas = ideas

    def __repr__(self):
        """represent the class's objects as a string"""
        return f'Person({self.apples},{self.ideas})'
```

```
if __name__ == '__main__':
    """ This is executed when run from the command line """
    main()
```

The code here effectively sums up the ideas of the crowd (there are only 2 people in our crowd, so how could you extend this to n people ?) and just swaps the apples.

The idea of tuple swapping is just another convenience that python offers compared to its predecessor languages. The equivalent c++ code requires the introduction of a temporary variable to perform the same swap which looks like this:

```
#include <iostream>
using namespace std;

int main()
{
    int a = 5, b = 10, temp;

    cout << "Before swapping." << endl;
    cout << "a = " << a << ", b = " << b << endl;

    temp = a;
    a = b;
    b = temp;

    cout << "\nAfter swapping." << endl;
    cout << "a = " << a << ", b = " << b << endl;

    return 0;
}
```

Python handles (even larger) swaps naturally.

```
a, b, c = 1, 2, 3
a, b, c = c, a, b

print(a,b,c)
```

We see two dunder methods in the Person Class. They are `__init__` and `__repr__`. We saw that `__init__` was for the initialization of the class since by default it is the first function that is run.

In Python, `__repr__` is a special method used to represent a class's objects as a string. `__repr__` is called by the `repr()` built-in function. You can define your own **string representation** of your class objects using the `__repr__` method.

## Rescue boats

An imminent hurricane threatens a coastal town.

*If at most two people can fit in a rescue boat, and the maximum weight limit for a given boat is  $k$ , determine how many boats will be needed to save everyone.*

For example, given a population with weights [100, 200, 150, 80] and a boat limit of 200, the smallest number of boats required will be three.

This is a typical fun question that could be solved with pen and paper for a small population size, but anything meaningful lends itself well in a programming language like python.

We want to look at all the combinations of two people from the sample set. If we can put the pairs (called tuples) into a list then we can determine all the tuples that satisfy the weight limit.

Python has a module named `itertools` which *standardises a core set of fast, memory efficient tools* and in particular for this question, deals with things like combinations and permutations.

We can also sort lists quickly. So in the example, the sorted list looks like this: [80, 100, 150, 200]. The nice thing about sorting the list before running the combinations is that the tuples are matched largest with smallest which just happens to be a feature that we desire (ie. matching two light people would be a waste of capacity, so we prefer to match heavy people with light ones).

And we can keep whittling the pairs of people with a recursion until there are no pairs left. So the solution will look like this:

```

import itertools as it
import time

# some population cases
population = [30, 100, 200, 150, 80, 40, 100, 100, 100, 100, 90]
population = [10, 10, 10, 10, 10, 180, 180, 180, 180, 180]
population = [100, 200, 150, 80]
boatLimit = 200

def rescue(population, boatLimit, numBoats):
    """
    Take population, limit and return smallest number of boats required
    input:
        population: array on numbers
        limit: number
    return:
        number: smallest number of boats
    """
    population.sort(reverse=True)
    c = it.combinations(population, 2)

    viablePairs = []
    for i in c:
        # find valid pairs
        if i[0]+i[1] <= boatLimit:
            viablePairs.append(i)

    print('reductions made:',viablePairs)

    if len(viablePairs)==0:
        #print('no viable pairs')
        numBoats = numBoats + len(population)
        print('the number of boats:',numBoats)

    else:
        # remove pair
        population.remove(viablePairs[0][0])

```

```

population.remove(viablePairs[0][1])
numBoats = numBoats + 1

# call function again
rescue(population, boatLimit, numBoats)

t1 = time.time()
rescue(population, boatLimit, 0)
t2 = time.time()
print('time taken:', round(t2-t1, 3), 'seconds')

```

- Again, can you do better?
- Or can you make a general form where more than 2 people, say  $n$  people, can fit into a boat?
- And how can we switch the recursion for a while loop ?

Just as a side note, we see that the number of boats needed is always greater than  $\text{len}(\text{population})/2$  because of the two person per boat rule. Moreover, for large random distributions, the solution approaches  $n/2$  because we can always find pairs of people to fit into the boats. So a statistical approach might be faster.

Remember, we can create a large random list with the `random` module and list comprehension very quickly like this.

```

import random
population = [random.randint(1,200) for x in range(1000)]

```

## Bracket Matching

This is an interesting little problem because the linter that you use will perform this check many times as you write code, which is extremely useful, but also something that we take for granted. So here it is:



*Given a string of round, curly, and square open and closing brackets, return whether the brackets are balanced (well-formed).*

*For example, given the string "([)][]({})", you should return true.*

*Given the string "([)]" or "((()", you should return false.*

```
openers = "{[("
closers = "})]"

# some strings for testing
inputstrings = ['()[]({}',
                '({})}',
                '[([]))}',
                '[]',
                '({',
                '()',
                '()'
                ]

# An opener is allowed anywhere but a closer must match the last opener

def parse(inputstring):
    # Work from left to right, appending and removing from stack
    # when encountering openers and closers respectively
    stack = []
    print(f"Input is {inputstring}")

    for pos, char in enumerate(inputstring):
        if char in openers:
            stack.append(char)

        elif char in closers:
            # closer found, check last item in stack is corresponding opener
            # and if so remove it from our stack
            if not stack:
                # stack is empty so no corresponding opener exists
                print(f"Closer '{char}' found without opener at pos {pos}")
```

```

    return False

    expected_closer = closers[openers.index(stack[-1])]
    if char == expected_closer:
        stack.pop() # all good
    else:
        # an error is found
        print(f' opener or "{expected_closer}" at pos {pos}, got "{char}"')
        return False

# End of string reached so stack should be empty
if stack:
    print(f'Expected opener or '{expected_closer}', got END OF STRING')
    return False
return True

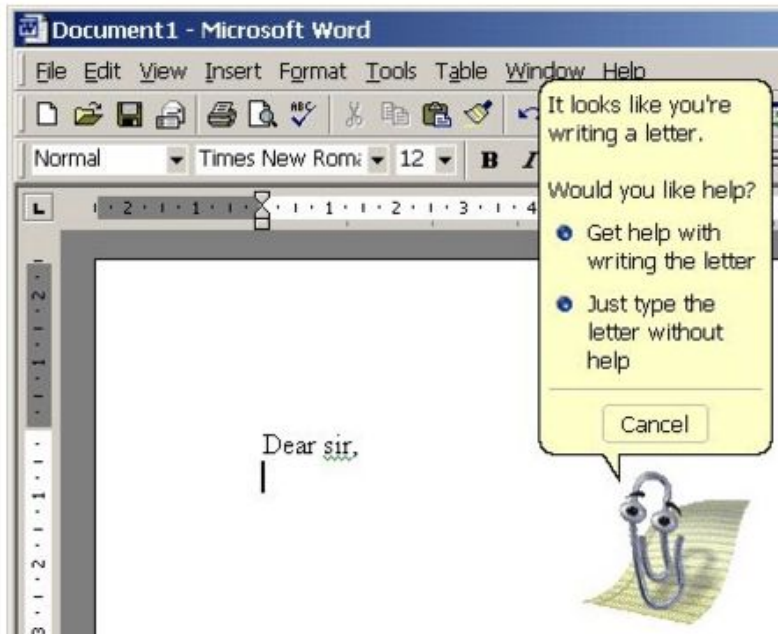
for i in inputstrings:
    print(parse(i))

```

The comments are done inline to save discussion, but the reader should be able to logically follow through at this stage.

The simple rule of rather basic bracket matching costs about 20 lines of code. When we consider all of the grammar, spelling and language functionality that we require it quickly becomes obvious that modules that handle “corrections” or “suggestions” must be sophisticated and have been in development for many years. For this, **we stand on the shoulders of giants**.

The older readers might remember the “paper clip” which was sometimes useful, but other times a hindrance.



There are many edge cases, for example a smiley face “ :) ” would create an additional bracket and throw an unmatched bracket error, whereas we actually wanted this.

```
# this is okay
print("hello world")

#Closer ')' found without preceding opener at pos 22
print("hello world :)")
```

So a good question for the reader is how would we overcome this? And other similar issues.

## Collatz Conjecture

The Collatz Conjecture is the simplest maths problem no one can solve. It is easy enough for almost anyone to understand but notoriously difficult to solve.

given any number:

- if odd:  $n = n * 3 + 1$

- if even:  $n = n/2$

The final value is always a [4,2,1] loop.

```
import matplotlib.pyplot as plt
from numpy import average, std

def f(n):
    """ the function itself:

    given an input n, what is the next value
    """
    if n % 2 == 0:
        # have an even number
        v = int(n/2)
        return v
    if n%2 !=0:
        # have an odd number
        v = n * 3 + 1
        return v

def number_iterations(n):
    """ given a number n, how many steps until a solution is reached

    """
    m = [n] # 4,2,1 loop

    for i in range(2):
        # first 3 elements of the loop
        n = f(n)
        m.append(n)
        # print(m)

    for i in range(2, 10000):
```

```

        # all the other loops
        n = f(n)
        # print(n)
        m.pop(0)
        m.append(n)
        # print(m)
        if m == [4,2,1]:
            return i

d = {}
for n in range(10_000):
    # n = 22 # initial number
    x = number_iterations(n)
    # print(n, x)
    d[n] = x

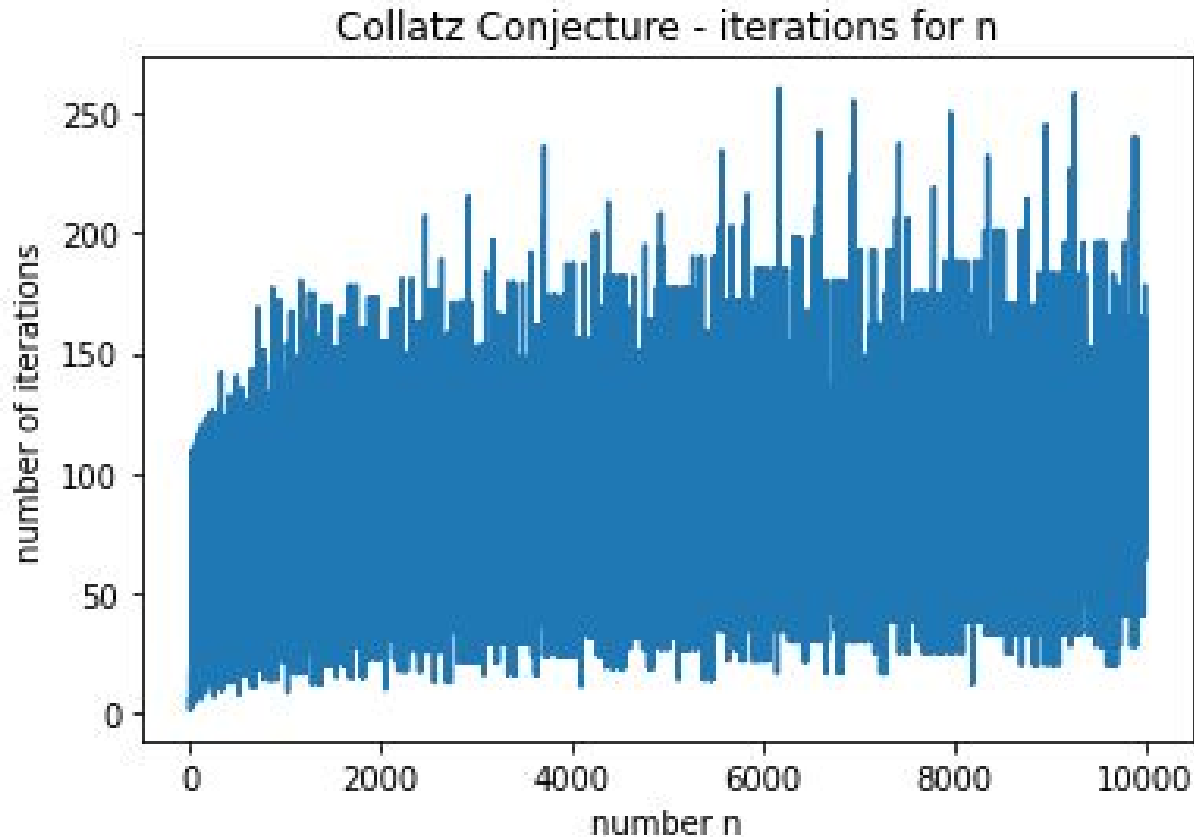
print(d.values())

# how many None's
print('number of nones', len([x for x in d.values() if x==None]) )
number_answers = [x for x in d.values() if x!=None]
print('max', max(number_answers))
print('avg', round(average(number_answers)))
print('std', round(std(number_answers)))

plt.plot(d.keys(), d.values())
plt.xlabel('number n')
plt.ylabel('number of iterations')

# giving a title to my graph
plt.title('Collatz Conjecture - iterations for n')
plt.show()

```



## Data Science

Data science combines maths and statistics, specialised programming, advanced analytics, artificial intelligence and machine learning to uncover insights hidden in data.

These insights can be used to guide decision making and strategic planning.

We often know what our end game is when it comes to starting a project, but along the way there are many steps.

- Data capture
- Data cleaning
- Processing
- More processing
- Producing coherent results

The challenges are enhanced at every stage usually because of [1] the size of the data and [2] the processing power and techniques available.

Python has many good tools for data science and it is for this reason that it has gained popularity in recent years. There is also a significant amount of material dedicated to data science specifically with python and the reader is encouraged to read this.

## Understanding Data

Data is the basis of everything that is done in this field. It is collected in many different ways, for example an image on a ccd array in a camera or a large database of trading data or a list or csv file that is collected via some research which might be a downloadable excel or csv file or published on a webpage.

This data collection is **sometimes easy** as it already comes in a convenient electronic homogeneous format (like all the trades on a particular stock on an exchange) and other times **rather difficult** like a population survey that is missing lots of data.

The data collection in itself is a programming issue as, however it is collected, there is a need to turn it into something electronic in order that we have a starting dataset to work with.

Let's take three similar large datasets that are collected in different ways.

1. Government Elections
2. Reality show votes
3. Social media polls

We say that the data is similar because it ultimately contains broadly speaking the same information. A unique person (the voter) and their choice (or vote).

The unique person might have some additional useful data, for example: postcode, annual income, age, gender etc. in fact, in python this would fit perfectly into a class structure or a dictionary or list .

```
class Voter:

    def __init__(self, vote, postcode, income, age, gender):
        self.vote = vote
        self.postcode = postcode
```

```
self.income = income
self.age = age
self.gender = gender

# voter instances
v1 = Voter('adam apple', 'abc', 35000, 32, 'male')
v2 = Voter('brenda banana', 'def', 25000, 28, 'female')

all_voters = {1000001:v1, 1000002:v2}
```

However, whilst the resultant collected data is the same, the collection methods vary widely.

**For the election**, the votes are collected via ballot papers which are either posted (and sent a few weeks prior to the election) or by voters going to private ballot boxes and filling out the ballot forms.

**For the reality shows**, the voters would already have been pre-registered on a website or mobile app or alternatively dial in by phone to cast their vote.

**For social media**, the voters all have accounts pre-registered setup and this would be the only way to vote.

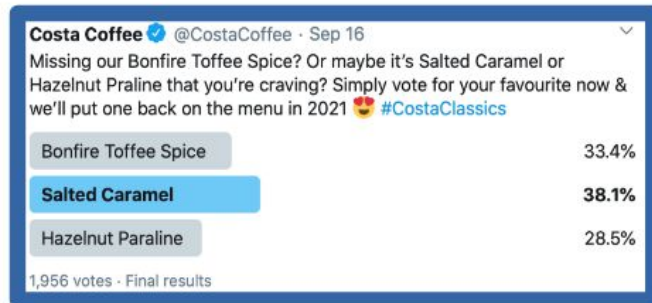
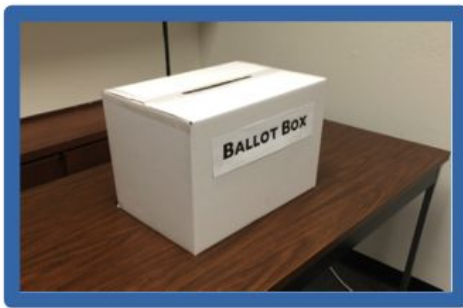
We see that *government election data is slow and clunky*. There is a lot of human effort and resources (paper and transport and manual reading) required to collate the information. Voters are not able to change their vote and those that did not vote properly or could not physically attend are excluded from the dataset. The turnout is around 65% meaning that 35% of eligible voters did not vote for a given reason. Nevertheless, the resulting dataset is large (30 million) as the original pool is roughly 45 million. The voting forms do not contain what one might consider to be rich data, but rather just a vote and an area.

On the other hand *reality show votes* have much richer data. As many of the users have pre-registered, a lot of information already existed. Aside from



the phone votes, the electronic votes are dynamic, meaning that voters can change their mind and this could even be real time.

And finally moving into *social media data* the information is even richer, often including far more detailed content such as personal or shopping habits with the results being real time.



Moving to other systems, the data is far more intensive. For example a self driving car will be continually collecting data via a number of sensors and feeding into a processing system for real time corrections and guidance.

In such a system, the efficiency of algorithms needs to be such that the available processing power is maximised. A car has seconds to react whereas an election cycle is 4 years which is approximately 100 million times the order or magnitude from a computational perspective. So we will need to consider all of the above.

### Data exploration: beginner

Data is contained on physical storage devices such as hard discs, usb drives and the internet. But it is also held in the computer's memory<sup>[3]</sup> for faster processing.

Python is used for accessing and processing both parts.

Let's start with a basic text file. How do we get this data from the hard disc into the computer RAM.

```
file_name = "C:/test/helloworld.txt"

with open(file_name) as f:
```

```
data = f.read()

print(data)
```

This reads the contents of the file into a variable that we named data. If the contents of the file are “hello world”, then:

```
data = 'hello world'
```

So python is quick and convenient at reading data. All the user needed was the file path and name . Note that we like to use the builtin ‘ with’ statement as this makes sure that the file is opened and closed properly.

Note the time taken to do this process is 1 to 2 milliseconds and a larger file size would take longer.

Just like we can read data, we can also write data.

```
file_name = "C:/test/helloworld.txt"

# create a string to write
write_string = ""
for i in range(1000):
    write_string = write_string + 'hello world\n'

# write the string
with open(file_name, 'w') as f:
    data = f.write(write_string)
```

On a mid range PC<sup>[4]</sup>, it was possible to write 100,000 lines of “hello world” or 2 megabytes of data in 15 milliseconds.

As the data gets larger, we need to consider writing chunks of data ( chunking ) in blocks and appending the file with each chunk so as to avoid memory issues. We can calibrate the chunk size and number of writes to optimise time.

```
# create a string to write
write_string = "
```

```

for i in range(10_000):
    write_string = write_string + ' ' + str(i) + ' hello world\n'

# write the first block
with open(file_name, 'w') as f:
    data = f.write(write_string)

# keep adding blocks (chunks)
for i in range(450):
    print(f'adding chunk {i+1}')
    with open(file_name, 'a') as f:
        data = f.write(write_string)

```

Again, on a mid range pc it was possible to write 45 million lines of “hello world” or 850 megabytes of data in just over 10 seconds. This was one “hello world” for each voter in the entire UK eligible electorate.

## Data exploration: intermediate

Having understood the fundamentals of data, where it is stored and how we can read and write something such that our code can now interact with this data it is time to move on to the popular modules. Namely Pandas; NumPy and SciPy

We were able to read and write basic data to a text file. We wrote “hello world” 45 million times quite quickly, which gave us an idea of speed and capability.

We now look at other data and what we can do.

The most basic way of displaying data is in simple flat file text format. This already provides the user with plenty of capability. The convention is to split the rows of a text file into a top row called the header and then each row below is an additional line of data.

When the data has structure, we split this into parts. Meaning that each row contains multiple pieces of data about one unique item. We could think of

this as a list of lists or a dictionary where the keys are the headings and the values are the lists of data. But let's just start simple and grow.

Imagine that we have this table of voters:

unique ref	first name	area	vote
100001	alice	West Midlands	conservative
100002	bob	London	conservative
100003	clive	Yorkshire	labour
100004	david	London	conservative
100005	emma	London	labour
100006	fred	Yorkshire	liberal democrat
100007	gill	South East	conservative
100008	harry	London	conservative
100009	irene	London	other

Each person is assigned a unique reference number so that they can only vote once. We have a first name, area and their vote. This would be sufficient for collating votes.

We represent this in a text file as follows:

```
unique ref,first name,area,vote
100001,alice,West Midlands,conservative
100002,bob,London,conservative
100003,clive,Yorkshire,labour
100004,david,London,conservative
100005,emma,London,labour
100006,fred,Yorkshire,liberal democrat
100007,gill,South East,conservative
100008,harry,London,conservative
100009,irene,London,other
```

What we have done is separate each column item with a comma. We could have used tabs, colons, semi-colons or | for the separation, but commas were sufficient. If the data had contained commas, we would have elected to use another method for separating the column data.

This format is known as comma separated values (another popular format is tab separated values) and has the file extension CSV. It is no more than a basic text file with rows of data that are separated with commas.

Python has a few ways of reading this data.

1. Text: read the text file and separate the data ourselves with a piece of code.
2. import csv: use the builtin module to handle the data
3. Import pandas: a powerful software library that handles data.

There are other methods too, but these are the most common.

The in build csv module that is shipped with python looks like this:

```
import csv

file_name = "C:/test/example_election.txt"

data = []
with open(file_name, newline="") as csvfile:
    spamreader = csv.reader(csvfile, delimiter=',')
    for row in spamreader:
        data.append(row)
```

And generates a list of lists which we have assigned to the variable data.

```
[['unique ref', 'first name', 'area', 'vote'],
['100001', 'alice', 'West Midlands', 'conservative'],
['100002', 'bob', 'London', 'conservative'],
['100003', 'clive', 'Yorkshire', 'labour'],
['100004', 'david', 'London', 'conservative'],
['100005', 'emma', 'London', 'labour'],
['100006', 'fred', 'Yorkshire', 'liberal democrat'],
['100007', 'gill', 'South East', 'conservative'],
['100008', 'harry', 'London', 'conservative'],
```

```
['100009', 'irene', 'London', 'other']]
```

But a more convenient tool is the `pandas`<sup>[5]</sup> module. This does all the work for the user in 2 lines of code.

```
import pandas as pd
```

```
df = pd.read_csv("C:/test/example_election.txt")
```

And pushes the result into an object called a `DataFrame` which looks like this.

	unique ref	first name	area	vote
0	100001	alice	West Midlands	conservative
1	100002	bob	London	conservative
2	100003	clive	Yorkshire	labour
3	100004	david	London	conservative
4	100005	emma	London	labour
5	100006	fred	Yorkshire	liberal democrat
6	100007	gill	South East	conservative
7	100008	harry	London	conservative
8	100009	irene	London	other

The `DataFrame` has a convenience (ie. fewer lines of code) advantage for small data, but the real advantage is when the data is large as the elements are contiguous and concise and can be operated on faster as a result (*especially when vectorised*<sup>[6]</sup>) in many, but not all, cases.

We can do things like filtering the columns (which are like lists) very conveniently.

```
x = df['vote']=='conservative' # filter conservatives
x.sum()      # count conservatives
# returns 5
```

So we can work out the percentage of conservative votes in just a few lines of code:

```
x = df['vote']=='conservative' # filter conservatives
count_conserv = x.sum()        # count conservatives
count_total = df['unique ref'].count() # total count

percent_conservative = round(count_conserv / count_total * 100, 2)
print('conservative percent is', percent_conservative, '%')
# conservative percent is 55.56 %
```

At this point we have already used Numpy without even knowing. The reason for this is that Pandas is built on top of NumPy, relying on ndarray and its fast and efficient array based mathematical functions. For example, when we did the count of the unique references we did this:

```
count_total = df['unique ref'].count() # total count
```

And if we wanted to find the mean we would have done `df['unique ref'].mean()`

The best description of numpy comes from the official website:

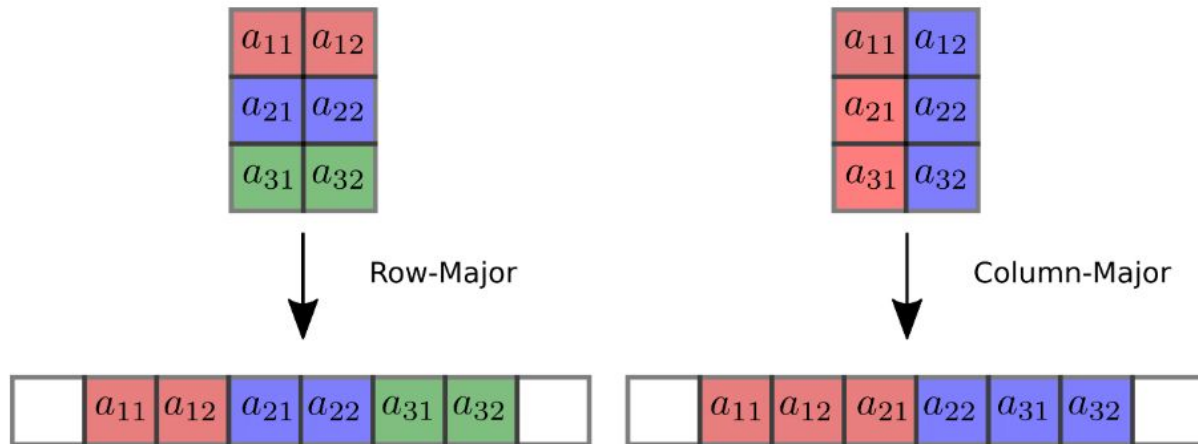
*NumPy is the fundamental package for scientific computing in Python. It is a Python library that provides a **multidimensional array object**<sup>[2]</sup>, various derived objects (such as masked arrays and matrices), and an assortment of **routines for fast operations on arrays**, including mathematical, logical, shape manipulation, sorting, selecting, I/O, discrete Fourier transforms, basic linear algebra, basic statistical operations, random simulation and much more.*

The power of NumPy comes from the ndarray class and how it is laid out in memory. The ndarray class consists of

1. The data type (all one type).
2. A pointer to a **contiguous** block of memory.
3. A tuple of the array shape.
4. An array stride.

The shape refers to the dimension of the array ( $a \times b$ ) while the stride is the number of bytes to step in a particular dimension when traversing an array in memory. With this NumPy has sufficient information to access all elements of the array.

NumPy also allows for **row** or **column** major, which can add to efficiency with the keyword `order`. A good summary looks like this:



The most important feature is the contiguous memory blocks compared to the standard python list. And also that each block has a single data type. The data could be of different data types (mixed types), but this would be at the expense of efficiency as the column becomes an object<sup>[8]</sup>. And the memory is now at multiple addresses.

We could delve into NumPy in much greater depth, but for the purpose of our needs of understanding the efficiency gains, this is sufficient. The NumPy ndarray is more efficient than the python list.

The most basic example is this:

```
import numpy as np
a = np.arange(15).reshape(3, 5)

print(a)
```

Which creates an array:

```
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14]])
```



We can get various details quickly.

```
a.shape      # (3, 5)
a.ndim       # 2
a.dtype.name  # 'int32'
a.itemsize   # 4 (4 bytes)
a.size       # 15
type(a)      # numpy.ndarray
```

And we can see that this looks like a “special” type of rigid list of integers with a specified size.

We can access the element of the ndarray like this:

```
print(a[1,1]) # return 6
print(a[1][1]) # returns 6
```

Although both of the above look the same, the first method is (approximately) twice as fast because it accesses the numpy API in C. Whereas method 2 returns a slice at [1] and then indexes the slice at [1] .

Finally, we look at scipy, which is another module. However, look no further than the strong relationship between SciPy and NumPy to the extent that the entire numpy namespace is included into scipy, which can be seen from this line in the SciPy `__init__` method.

```
from numpy import *
```

SciPy is built upon NumPy.

## Data visualisation

Matplotlib; Seaborn;

Up to this point, we have looked at data crunching. We could transform and output data via the terminal or a pandas dataframe or write to a file.

However, up to this point, we had not done any kind of visualisation. So let's do this. And we start with Matplotlib.

Matplotlib is a comprehensive library for creating static, animated, and interactive visualisations in Python. Matplotlib makes easy things easy and

hard things possible.

We will often see Matplotlib come hand in hand with Numpy Pandas or Scipy. This is because, yet again, like the other modules Matplotlib leverages the same Numpy ndarray which is fast.

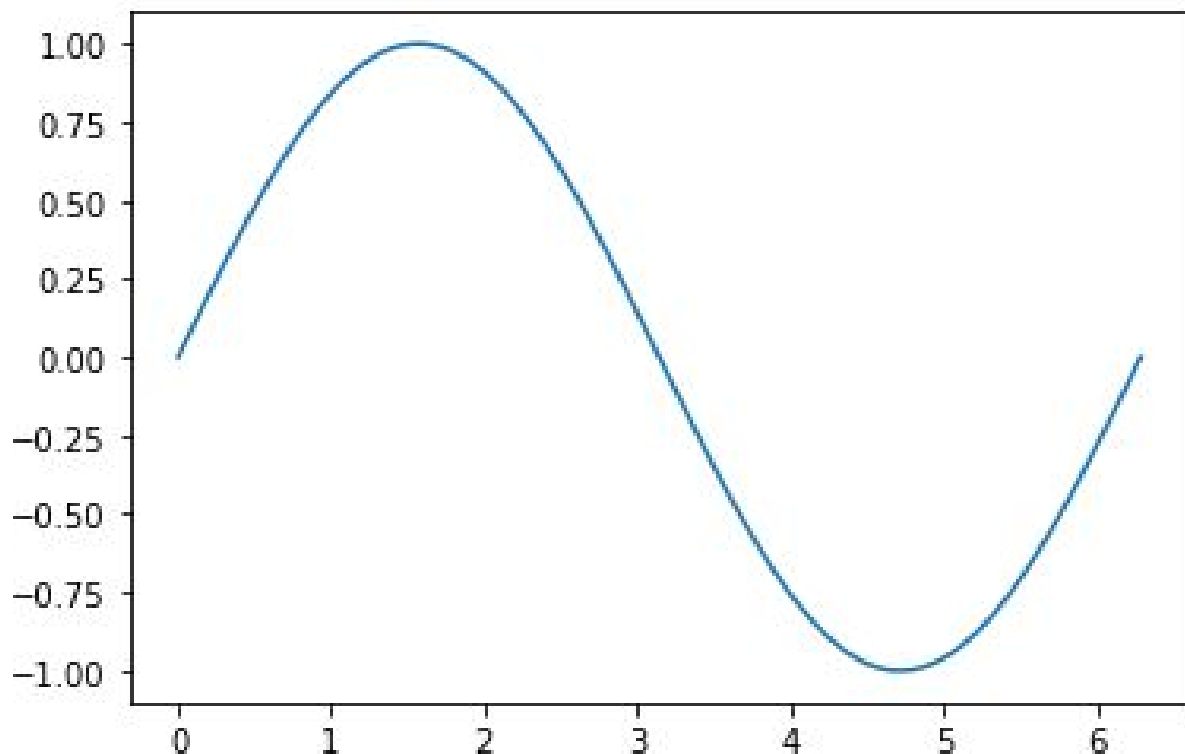
The basic example, just to start quickly, is this:

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

x = np.linspace(0, 2 * np.pi, 200)
y = np.sin(x)

fig, ax = plt.subplots()
ax.plot(x, y)
plt.show()
```

Which quickly produces a sine wave and plots the result conveniently into a chart.



We can change the plot type, like this:

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

# make data:
np.random.seed(3)
x = 0.5 + np.arange(8)
y = np.random.uniform(2, 7, len(x))

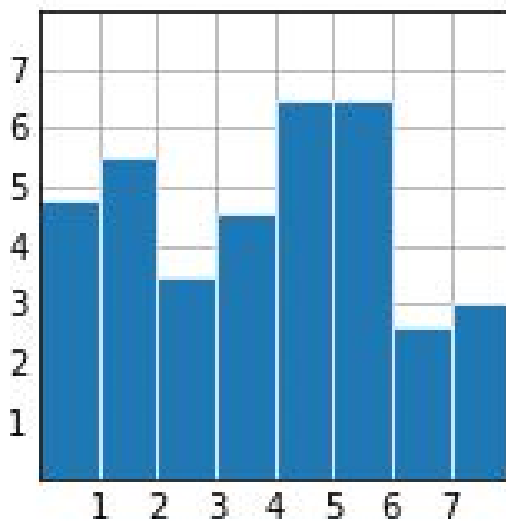
# plot
fig, ax = plt.subplots()

ax.bar(x, y, width=1, edgecolor="white", linewidth=0.7)

ax.set(xlim=(0, 8), xticks=np.arange(1, 8),
       ylim=(0, 8), yticks=np.arange(1, 8))

plt.show()
```

Which gives this:



And more complicated data like this:

```

import matplotlib.pyplot as plt
import numpy as np

# make data:
np.random.seed(1)
x = np.random.uniform(-3, 3, 256)
y = np.random.uniform(-3, 3, 256)
z = (1 - x/2 + x**5 + y**3) * np.exp(-x**2 - y**2)
levels = np.linspace(z.min(), z.max(), 7)

# plot:
fig, ax = plt.subplots()

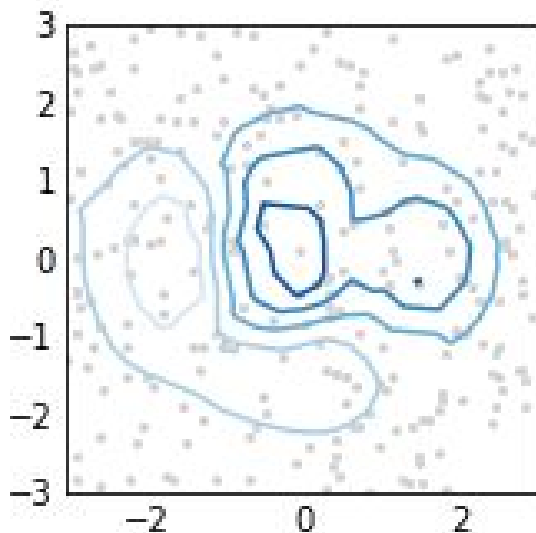
ax.plot(x, y, 'o', markersize=2, color='lightgrey')
ax.tricontour(x, y, z, levels=levels)

ax.set(xlim=(-3, 3), ylim=(-3, 3))

plt.show()

```

Which produces this:



We continue, noting that the list of plots is extensive, so only need to go as far as to know that the **options are vast**<sup>[9]</sup> and that the structure of the code to produce a plot is roughly 10 to 20 lines of code and is similar each time.

- Get or create the data
- Organise the data and how the chart looks
- Plot the chart (with `plt.show()` )

We can now begin to do statistical analysis. Imagine that we have x and y points and we want to find a simple relationship and draw a best fit line. Combining numpy with matplotlib this can easily be achieved.

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

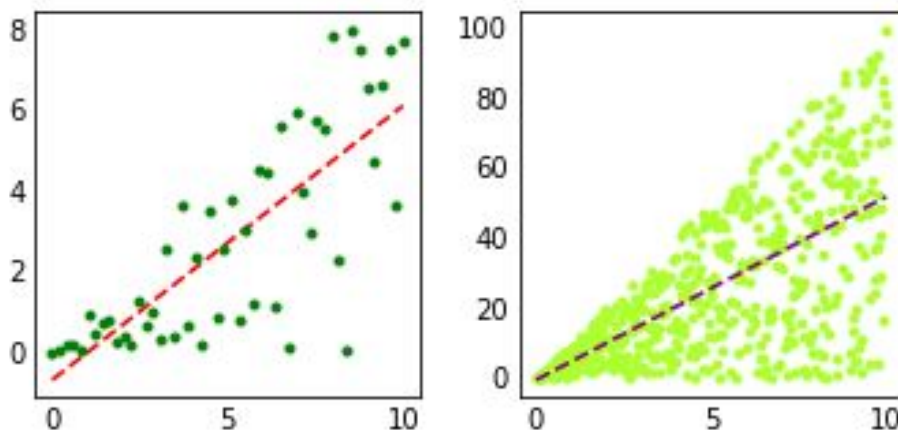
# create x and y points
x = np.linspace(0, 10, 50)
y = np.random.random(50) * np.linspace(0,10,50)

# fit a linear curve an estimate its y-values and their error.
a, b = np.polyfit(x, y, deg=1)
y_est = a * x + b

fig, ax = plt.subplots()
ax.plot(x, y, '.', color='green')
ax.plot(x, y_est, '--', color='red')

plt.show()
```

This is the result for 50 and 500 points respectively:



We were able to quickly generate and plot many points and produce **lines of best fit** repeatedly (tweaking parameters and colours along the way).

We used one function for the curve (or straight line) fitting, which is `polyfit(x,y, deg=1)` . If we had used `deg=2` then we would have had a quadratic and `deg=3` for a cubic fit and so forth. And this was all that we needed. The results are unpacked<sup>[10]</sup> into the coefficients.

This is the beginning of a long journey into data science and what tools and techniques we have to describe, interpret and even predict data.

Or become a bit smarter with standard deviations from a fit:

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

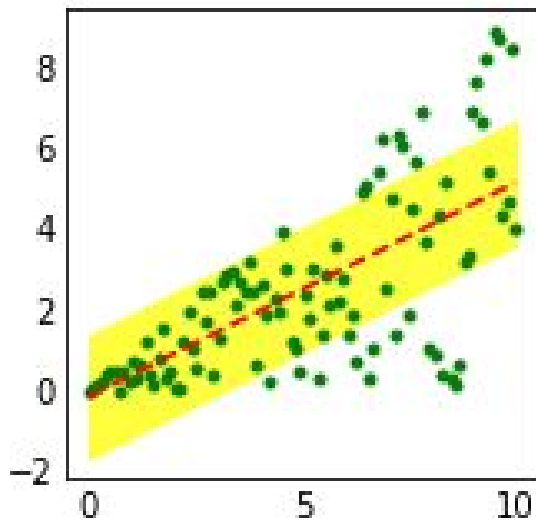
# create x and y points
n = 100
x = np.linspace(0, 10, n)
y = np.random.random(n) * np.linspace(0,10,n)

# fit a linear curve an estimate its y-values and their error.
a, b = np.polyfit(x, y, deg=1)
y_est = a * x + b
y_err = np.sqrt(y.std()) # an error estimator

fig, ax = plt.subplots()
ax.fill_between(x, y_est - y_err, y_est + y_err, color='yellow', alpha=0.75)
ax.plot(x, y, '.', color='green')
ax.plot(x, y_est, '--', color='red')

plt.show()
```

Which gives this:



A good takeaway is that this is achieved relatively quickly using python and has use cases in all industries, like finance, engineering, retail, academia and so forth.

We could have looked at surfaces, 3 dimensions, n-dimensions animations, so there is plenty of scope for the reader to extend their knowledge.

### Machine learning

We could define this as any supervised or unsupervised learning task (that is not deep learning). Scikit-learn is the main go-to tool module for implementing classification, regression, clustering and so forth (although other modules also exist, like OpenCV and TensorFlow).

Machine learning utilises statistical techniques to give computer algorithms the ability to learn from past experiences and perform specific tasks.

So we touched on data visualisation. We were able to display charts from various basic statistics. We used a best fit line on a linear random distribution, but there are many other fits and distributions that we could have just as easily deployed.

We multiplied numpy arrays (the dot product for vectors) with ease:

```
y = np.random.random(n) * np.linspace(0,10,n)
```

To create the independent y-variable and could have easily added a log normal distribution or other non linear distribution.

```

import matplotlib.pyplot as plt
import numpy as np

# create x and y points
n = 100
x = np.linspace(0, 10, n)
y = np.random.random(n) * x * 100

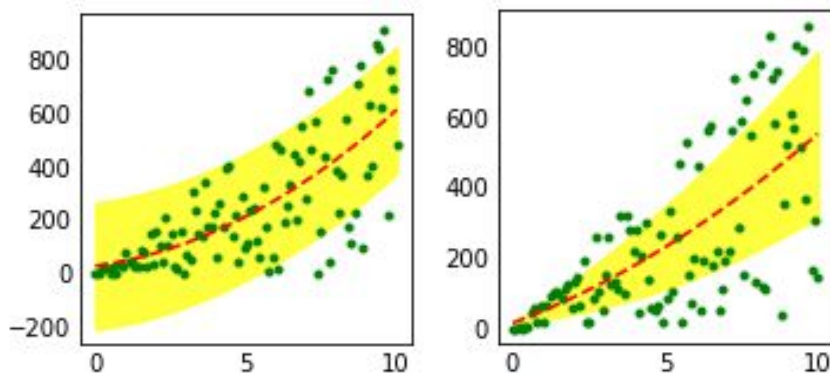
# fit a linear curve an estimate its y-values and their error.
a, b, c = np.polyfit(x, y, deg=2)
y_est = (a * x * x) + (b * x) + c
y_err = y.std()

fig, ax = plt.subplots()
ax.fill_between(x, y_est - y_err, y_est + y_err, color='yellow', alpha=0.75)
ax.plot(x, y, '.', color='green')
ax.plot(x, y_est, '--', color='red')

plt.show()

```

To get this:



We can create distributions that one fits to the data. Common ones are normal and lognormal distributions. Which are generated like this:

```

import math
import numpy as np
from scipy.stats import lognorm

```



```

import matplotlib.pyplot as plt

#make this example reproducible
np.random.seed(1)

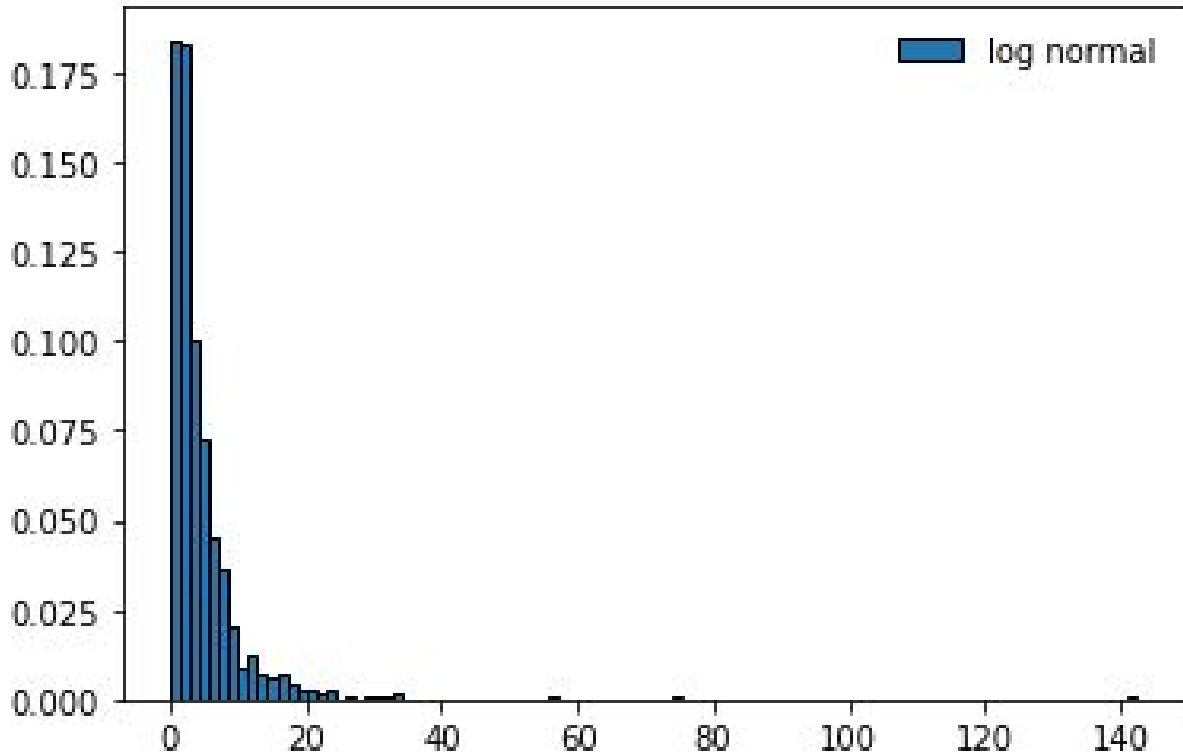
#generate log-normal distributed random variable with 1000 values
avg, std = 1, 1
lognorm_values = lognorm.rvs(s=std, scale=math.exp(avg), size=1000)

#view first five values
lognorm_values[:5]

#create histogram
plt.hist(lognorm_values, density=True, edgecolor='black', bins=100, label='log normal')
plt.legend(loc='best', frameon=False)
plt.show()

```

And produce this



We can look at statistics like mean, standard deviation, quartiles, confidence levels etc. The data will exist in the main body or it could be classified as an outlier. For example more than 3 standard deviations from the dataset mean, which is a regression type model.

We could also have a decision tree type model. We saw the basics of this in the binary tree where each node of the tree is a decision and more nodes generally leads to greater accuracy. The sklearn module in python gives us a whole suite of classification tools. Here for example is a decision tree classifier on a popular dataset called iris<sup>[11]</sup> out of the box.

```
from sklearn.datasets import load_iris
from sklearn.model_selection import cross_val_score
from sklearn.tree import DecisionTreeClassifier
clf = DecisionTreeClassifier(random_state=0)
iris = load_iris()
cross_val_score(clf, iris.data, iris.target, cv=10)
```

Which returns this list

```
array([1.      , 0.93333333, 1.      , 0.93333333, 0.93333333,
        0.86666667, 0.93333333, 1.      , 1.      , 1.      ])
```

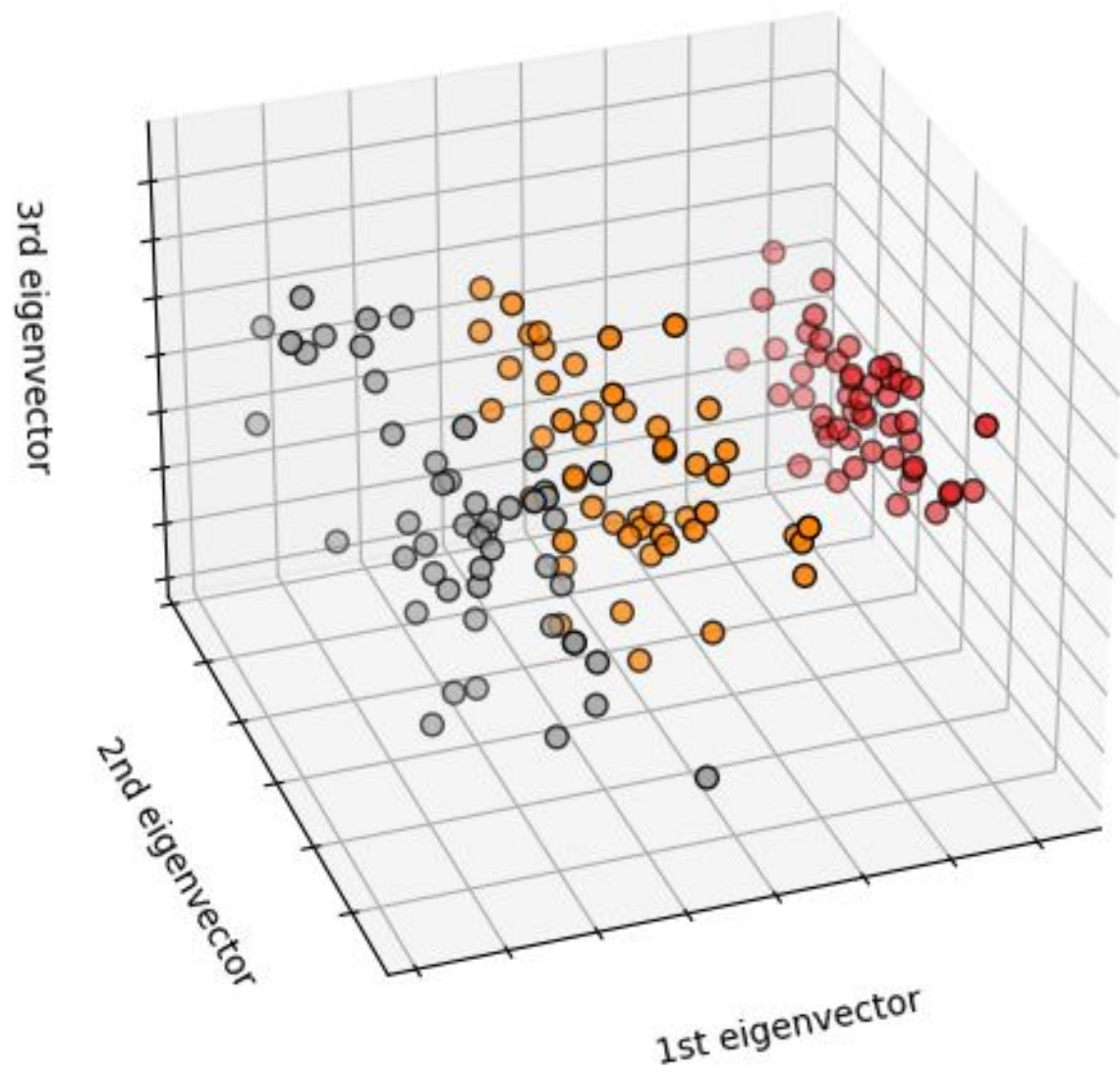
The number of inputs (parameters) that could go into the DecisionTreeClassifier model reflect the potential complexity and this is one of many models. So what we have just touched the surface.

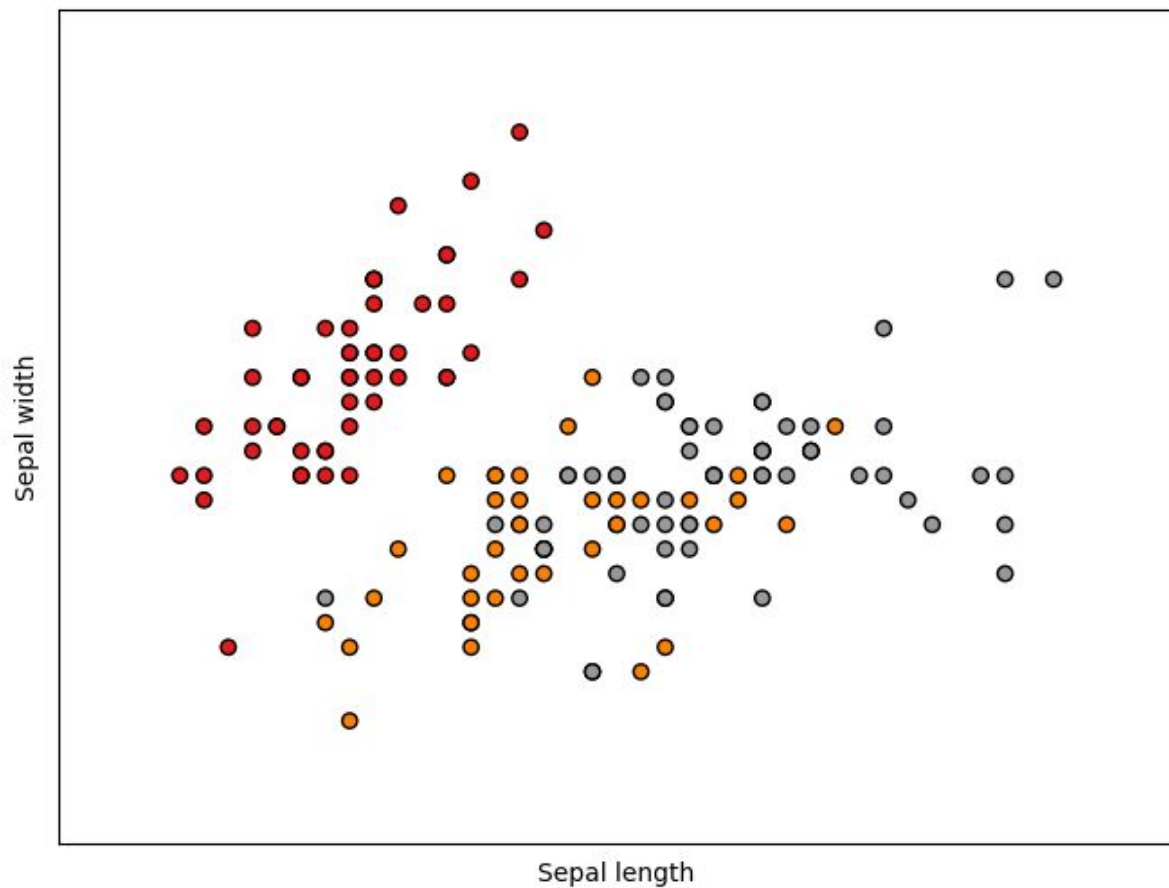
The documentation allows for the following:

```
class sklearn.tree.DecisionTreeClassifier(*, criterion='gini', splitter='best', max_depth=None,
min_samples_split=2, min_samples_leaf=1, min_weight_fraction_leaf=0.0, max_features=None,
random_state=None, max_leaf_nodes=None, min_impurity_decrease=0.0, class_weight=None,
ccp_alpha=0.0)
```

And the scope of just this problem alone would occupy the rest of the book. However, the important thing to note is that the (Iris Dataset) data looks like this:

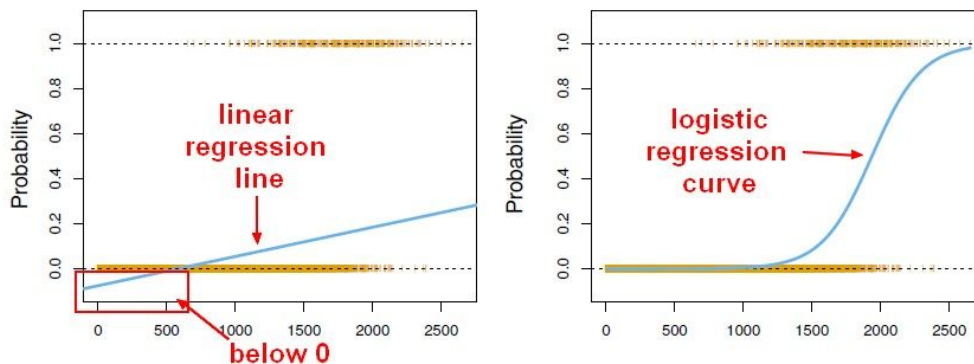
First three PCA directions





And importantly the model is classifying the type of flower based on the existing information. For example, given a short sepal length and high sepal width the model would expect a flower whose type rests among red data points.

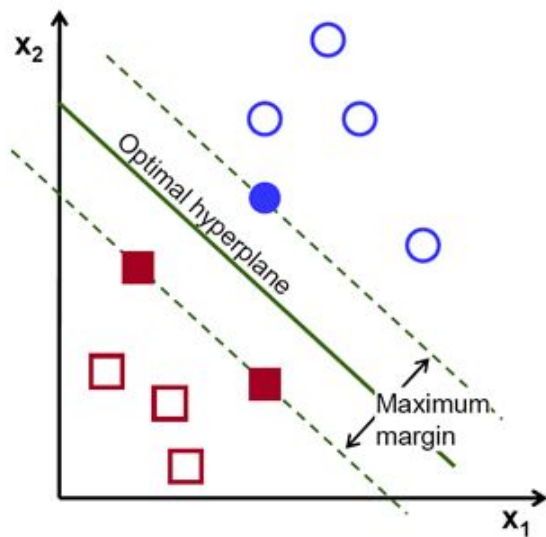
We have shifted into the realms of logistic regression which is similar, but not to be confused with linear regression. Logistic regression is used to model the probability of a finite number of outcomes, typically two.



In essence, a logistic equation is created in such a way that the output values can only be between 0 and 1.

With this we now look at the Support Vector Machine (SVM) which is another supervised classification technique.

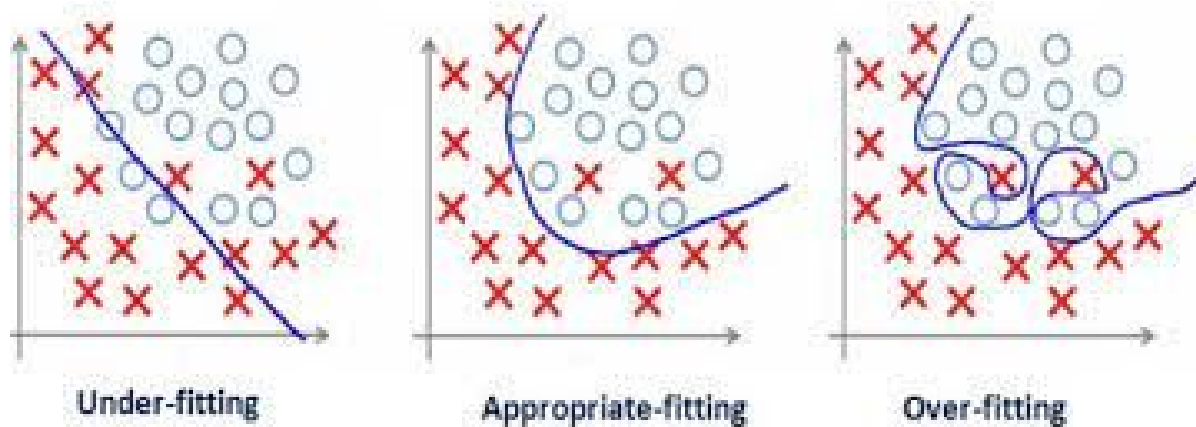
Given two classes of data. A SVM will find a hyperplane or a boundary between the two classes of data that maximises the margin between the two classes.



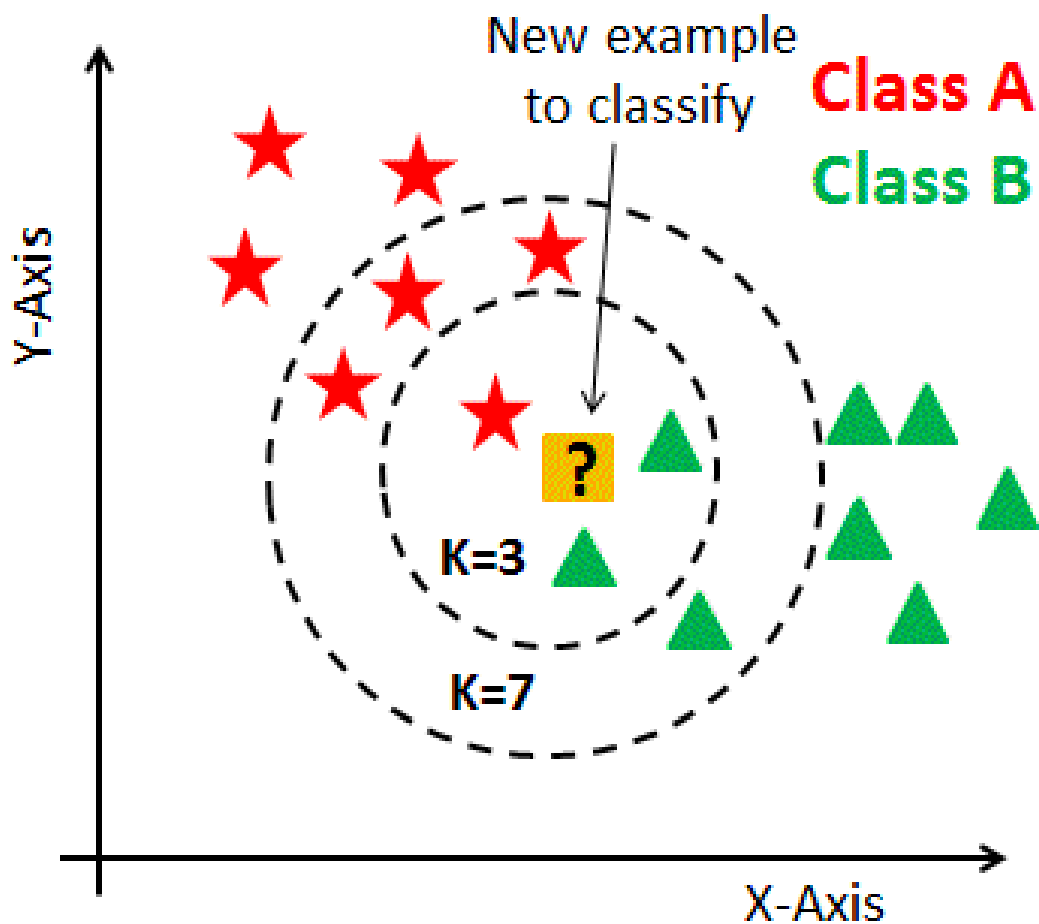
There are many planes that can separate the two classes, but we seek a single plane that maximises the margin (or distance) between the classes which are the blue and red points in the diagram.

Unlike supervised learning, unsupervised learning is used to draw inferences and find patterns from input data without references to labelled outcomes.

One such method is clustering that involves the grouping, or clustering, of data points. We had begun to touch on this, but let's take a deeper dive using the sklearn knn<sup>[12]</sup> classifier called the KNeighborsClassifier .



Whichever technique is used, the task is essentially to classify data as best as possible.



KNN is a very interesting technique. K is the number of nearest neighbours and this is the core deciding factor. With a simple counting algo, it helps if k is an odd number given that we do not want a split decision.

There are different and interesting measures for finding the distances such as:

- Euclidean distance
- Hamming distance
- Manhattan distance
- Minkowski distance

But essentially each one attempts to achieve the same outcome.

Techniques like KNN perform better with a lower number of features. When the number of features increases, the amount of data required also increases. This is known as the curse of dimensionality.

And finally, how do we even decide on the number of nearest neighbours? These are all considerations and therefore input arguments into the models that we see.

All of the above shows the complexity that goes into the algo's, but we are here for the python implementation and some use cases and examples.

```
X = [[0], [1], [2], [3]]
y = [0, 0, 1, 1]

from sklearn.neighbors import KNeighborsClassifier
neigh = KNeighborsClassifier(n_neighbors=3)
neigh.fit(X, y)

print(neigh.predict([[1.1]]))

print(neigh.predict_proba([[1.1]]))
```

So let's start with this example using the KNeighborsClassifier method which returns this:

```
[0]
```

```
[[0.66666667 0.33333333]]
```

Basically the model returns zero for the given x,y data in the top two rows of the example code and also the probabilities of such (67% vs 33%) remembering that the probability space always adds to 100%.

In this next example we construct a `NearestNeighbors` class from an array representing our data set and ask who's the closest point to [1,1,1].

```
samples = [[0., 0., 0.], [0., .5, 0.], [1., 1., .5]]
from sklearn.neighbors import NearestNeighbors
neigh = NearestNeighbors(n_neighbors=1)
neigh.fit(samples)

print(neigh.kneighbors([[1., 1., 1.]])
```

Which returns this:

```
(array([[0.5]]), array([[2]], dtype=int64))
```

Telling us that the “distance” is 0.5 and it was the third element in the samples list (remembering that indexes start from zero).

At the end of all of the theory, it is nice to show good code. And here is a comparison of 10 classifiers in scikit-learn. The point of this example is to illustrate the nature of decision boundaries of different classifiers.

```
# standard imports
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.colors import ListedColormap

# import models
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.datasets import make_moons, make_circles, make_classification

# import the classifiers
from sklearn.neural_network import MLPClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.svm import SVC
from sklearn.gaussian_process import GaussianProcessClassifier
```



```

from sklearn.gaussian_process.kernels import RBF
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier, AdaBoostClassifier
from sklearn.naive_bayes import GaussianNB
from sklearn.discriminant_analysis import QuadraticDiscriminantAnalysis
from sklearn.inspection import DecisionBoundaryDisplay

names = [ # names of classifiers
    "Nearest Neighbors",
    "Linear SVM",
    "RBF SVM",
    "Gaussian Process",
    "Decision Tree",
    "Random Forest",
    "Neural Net",
    "AdaBoost",
    "Naive Bayes",
    "QDA",
]

classifiers = [ # models
    KNeighborsClassifier(3),
    SVC(kernel="linear", C=0.025),
    SVC(gamma=2, C=1),
    GaussianProcessClassifier(1.0 * RBF(1.0)),
    DecisionTreeClassifier(max_depth=5),
    RandomForestClassifier(max_depth=5, n_estimators=10, max_features=1),
    MLPClassifier(alpha=1, max_iter=1000),
    AdaBoostClassifier(),
    GaussianNB(),
    QuadraticDiscriminantAnalysis(),
]

X, y = make_classification(
    n_features=2, n_redundant=0, n_informative=2, random_state=1, n_clusters_per_class=1
)

rng = np.random.RandomState(2)

```

```

X += 2 * rng.uniform(size=X.shape)
linearly_separable = (X, y)

datasets = [
    make_moons(noise=0.3, random_state=0),
    make_circles(noise=0.2, factor=0.5, random_state=1),
    linearly_separable,
]

figure = plt.figure(figsize=(27, 9))
i = 1
# iterate over datasets
for ds_cnt, ds in enumerate(datasets):
    # preprocess dataset, split into training and test part
    X, y = ds
    X = StandardScaler().fit_transform(X)
    X_train, X_test, y_train, y_test = train_test_split(
        X, y, test_size=0.4, random_state=42
    )

    x_min, x_max = X[:, 0].min() - 0.5, X[:, 0].max() + 0.5
    y_min, y_max = X[:, 1].min() - 0.5, X[:, 1].max() + 0.5

    # just plot the dataset first
    cm = plt.cm.RdBu
    cm_bright = ListedColormap(["#FF0000", "#0000FF"])
    ax = plt.subplot(len(datasets), len(classifiers) + 1, i)
    if ds_cnt == 0:
        ax.set_title("Input data")
    # Plot the training points
    ax.scatter(X_train[:, 0], X_train[:, 1], c=y_train, cmap=cm_bright, edgecolors="k")
    # Plot the testing points
    ax.scatter(
        X_test[:, 0], X_test[:, 1], c=y_test, cmap=cm_bright, alpha=0.6, edgecolors="k"
    )
    ax.set_xlim(x_min, x_max)
    ax.set_ylim(y_min, y_max)

```

```

ax.set_xticks(())
ax.set_yticks(())

i += 1

# iterate over classifiers
for name, clf in zip(names, classifiers):

    ax = plt.subplot(len(datasets), len(classifiers) + 1, i)
    clf.fit(X_train, y_train)
    score = clf.score(X_test, y_test)
    DecisionBoundaryDisplay.from_estimator(
        clf, X, cmap=cm, alpha=0.8, ax=ax, eps=0.5
    )

    # Plot the training points
    ax.scatter(
        X_train[:, 0], X_train[:, 1], c=y_train, cmap=cm_bright, edgecolors="k"
    )

    # Plot the testing points
    ax.scatter(
        X_test[:, 0],
        X_test[:, 1],
        c=y_test,
        cmap=cm_bright,
        edgecolors="k",
        alpha=0.6,
    )

    ax.set_xlim(x_min, x_max)
    ax.set_ylim(y_min, y_max)
    ax.set_xticks(())
    ax.set_yticks(())
    if ds_cnt == 0:
        ax.set_title(name)
    ax.text(
        x_max - 0.3,
        y_min + 0.3,
        ("%.2f" % score).lstrip("0"),

```

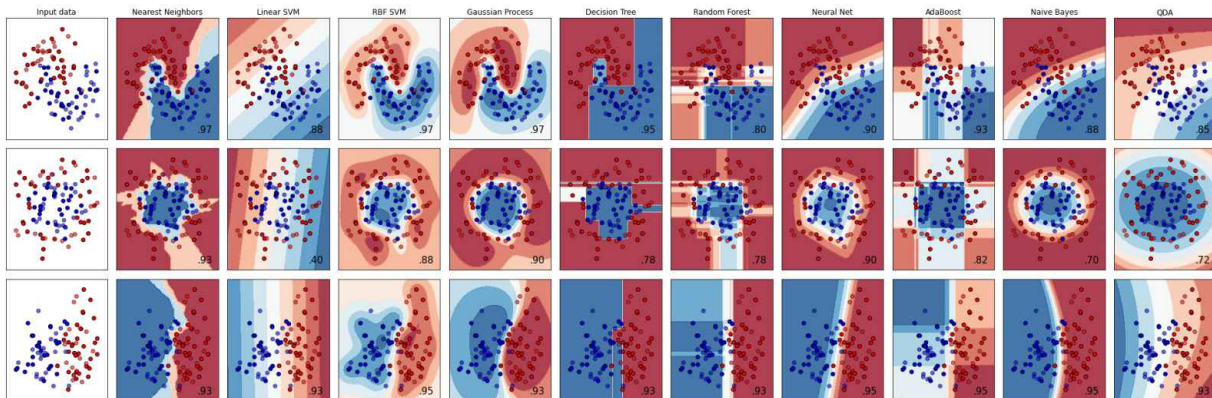
```

        size=15,
        horizontalalignment="right",
    )
    i += 1

plt.tight_layout()
plt.show()

```

This produces the following (in less than 3 seconds):



The full code can be found on the scikit-learn website. The plots show training points in solid colours and testing points semi-transparent. The lower right shows the classification accuracy on the test set.

What we can garnish from the above is that, just in one section of machine learning alone the computation volume is immense and this is before we have transposed ideas onto real world data which is significant.

## Finance

Python is a *general purpose high level programming language* which lends itself very well to the finance sector primarily because it bridges a very specific gap somewhere between spreadsheets and lower level languages.

type	example	difficulty	computation	Build time
Spreadsheet (code)	excel	easy	slow	fast
High level code	python	medium	medium	medium
Low level code	c++	hard	fast	fast

The syntax is clean and dealing with lists and arrays is relatively straight forward. The language connects well with other parts of the system. Accessing databases and reading from and writing to files is easy.

There is a level of automation in python that is beyond that of excel and it is simpler to use than (more powerful) languages like c++ that have longer development times.

So python is good for prototyping. For example, let's take a look at the lognormal distribution which is one of the most common distributions in finance or infact any biological system whose distribution has a lower bound at zero and is random in nature.

- Take a bunch of random variables.
- Take the log of those:  $Y = \ln(x)$

We know that the dependent variable, y is some function of x and in the case of the lognormal distribution some function of the natural logarithm of x.

$$y = f(x) \Leftrightarrow x = f^{-1}(y)$$

$$y = \ln(x) \Leftrightarrow x = e^y$$

Here we set the mean,  $\mu$ , and standard deviation  $\sigma$ .

$$y = \mu + \sigma x$$

If  $x$  is normal, then  $\mu + \sigma x$  is also normal as the transformations just scale the distribution, and do not affect normality, which means that the logarithm of  $x$  is normally distributed.

The lognormal distribution satisfies this:

$$f(x) = \frac{1}{x\sigma\sqrt{2\pi}} e^{-\left(\frac{(\ln x - \mu)^2}{2\sigma^2}\right)}$$

This is easy to replicate in python.

```
import math
import numpy as np
import statistics as sts
from matplotlib import pyplot as plt

n = 1000          # number of samples
avg, std = 100, 0.2 # avg , std
mu = math.log(avg)
sigma = math.sqrt(std)
y = np.random.lognormal(mean=mu, sigma=sigma, size=n)
```

```

print('avg:', round(sts.mean(y), 2))
print('std:', round(sts.stdev(y), 2))

fig, ax = plt.subplots(1,2)

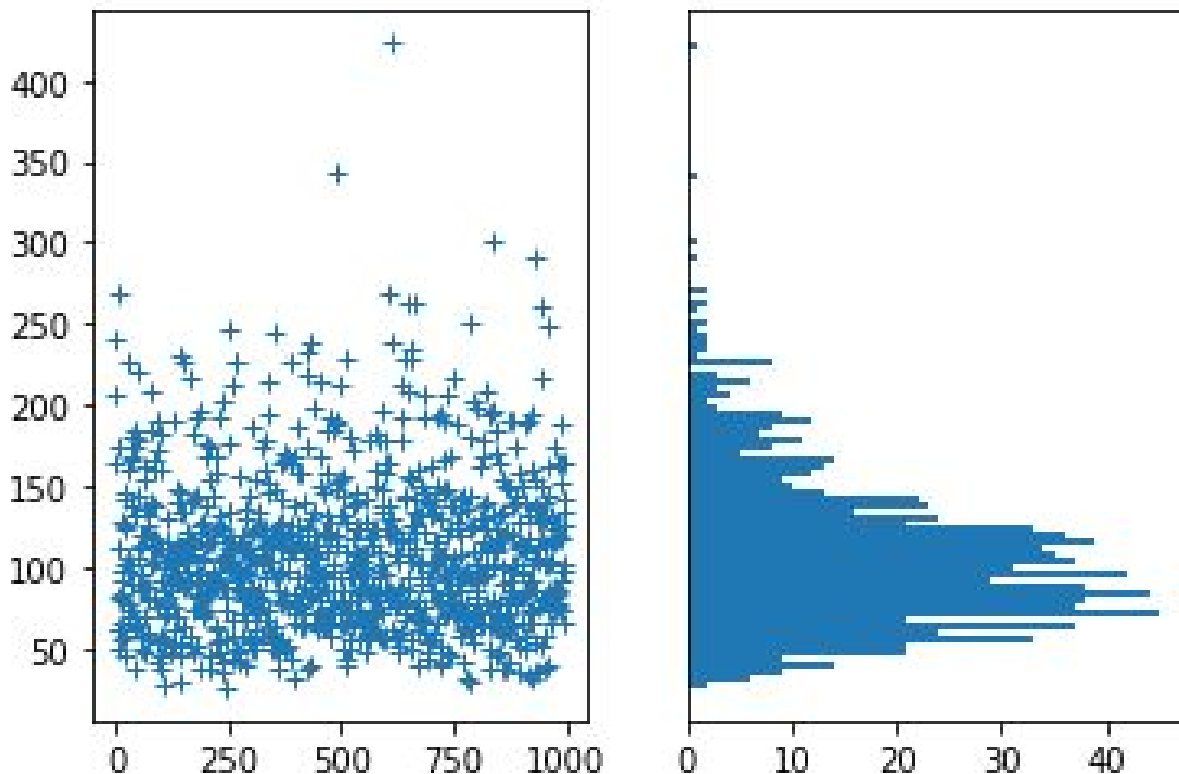
ax[0].plot(y, '+')
# plt.show()

ax[1].hist(y, bins=int(n/10), orientation='horizontal')
ax[1].set_yticks([])
plt.show()

```

The imports at the top provide tools to do all the basic maths operations and the entire distribution is generated by the `np.random.lognormal()` function.

We are able to plot this using matplotlib showing 1000 points, but also plotting a histogram oriented sideways to indicate the frequency of the plots.



For this particular chart the average was 108.36 and std of 47.59. We could run this routine many times, which would create many similarly looking random distributions.

Running simulations many times is a common technique especially with the increased computing power that is available and is referred to as a monte-carlo simulation and takes a number of different formats.

## Time series

The time series is important because it represents the evolution of a function (price) with respect to time. In finance, other value metrics are used, such as bond yields or spreads (which are the differences in yields).

We often see prices dropping and yields rising or spreads widening when the market is taking off risk and conversely prices rising and yields dropping or spreads tightening when the market is adding risk.

We can capture this data and measure and analyse it and make various statistical statements about the systems that we have measured.

There are a few approaches, but let's try these two:

- generate our own sample data by creating a random walk
- we could download a common dataset.

The random walk is fun and we can do lots with it, plus it is good for testing out some of the techniques that we have been through and has real world applications.

In the simplest case, we start with a price and make it evolve over discrete intervals of time, such as days. So we can say that the price tomorrow is some function of the price today or equivalently the next price was contingent on the previous price and some other input parameters that the user sets.

$$P_{n+1} = f(P_n)$$

For the price of the stock, we might crudely assume that there is an upward drift and on top of this we could add a random value to represent an unknown volatility component, so we have this:



$$P_{n+1} = (\alpha + \beta_n)P_n$$

Where  $\alpha$  is a predefined constant and  $\beta_n$  is a random number generated at each step. So we have a **drift** and a **diffusion**.

Let's say that over the course of a year, the price should increase by 5%, with no volatility (so,  $\beta_n = 0, \forall n$ ). Then the following would be true:

$$P_{365} = \alpha^{365}P_0$$

And therefore the term  $\alpha^{365}$  is equal to 1.05. So we can solve for  $\alpha$ .

$$\alpha = 1.05^{1/365}$$

And if we introduce a suitable volatility, say  $\beta = \frac{\text{random}(0,1)}{100}$ , then it becomes possible to attempt reasonable walks.

Again, we produce this in 30 lines of code.

```
import random
from matplotlib import pyplot as plt

start_price = 100
number_of_steps = 365

# 5% over a 1-year period
drift = pow(1.05, 1/365)

def random_walk():
    """ a random walk function"""
    p = [start_price]
    for i in range(number_of_steps):
        vola = (random.random() - 0.5)/100
        p_next = (drift + vola) * p[i]
        p.append(p_next)
```

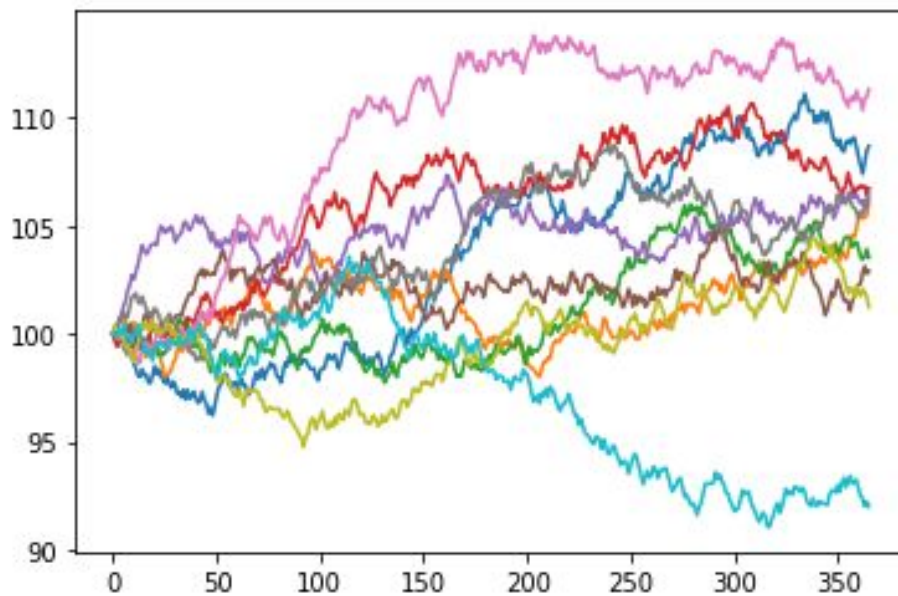
```
    return p

# lets do 100 random walks (list comprehension)
walks = [random_walk() for x in range(100)]

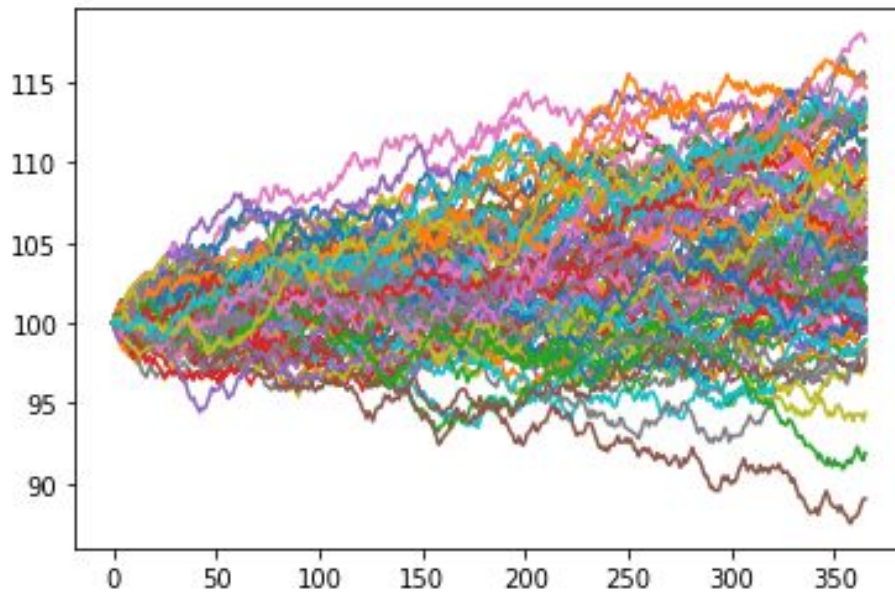
# plot all the paths
for i in walks:
    plt.plot(i)

plt.show()
```

This is what 10 random walks each of 365 daily steps over the course of 1 year looks like with the model above.



And this is what 100 walks look like:



We could have simulated many thousands of walks with multivarious outcomes based on sophisticated parameters and it would be quite common to see this in other disciplines such as weather forecasting or engineering problems where the models are complex and intensive.

In particular, if we consider a weather system evolving over time, that the 1-dimensional time steps that we used in the example become 3 spatial dimensions evolving over time (so 4 dimensions) and all of a sudden our 365 calculations grows to  $365^3$  (or roughly 50 million calculations).

If we were to take a weather system over the uk which is roughly 240,000 km<sup>2</sup> and consider cubes of 1km then the model would require 250 cubes just for the basic footprint. And if we wanted to increase the accuracy to 1 metre blocks, this would quickly bifurcate to 240 million volumes for a single iteration of evolution. All of a sudden the computational systems are challenged.

Next we look at a common dataset, this could be any, but let's take the ftse 100, which is an index that tracks the top 100 shares by market capital in the UK.

We have 25 years worth of data which is roughly 6250 data points assuming 250 working days per year. There are 4 data points on each day [open, high, low, close].

The data comes in two downloadable formats. A CSV file which we have already discussed in earlier chapters and a JSON<sup>[13]</sup> file.

With either format, we can quickly download the data into a pandas dataframe with the `pandas.read_csv()` or `pandas.read_json()` methods.

An extract of the json format looks like this:

```
[{"Symbol": "UKX:IND", "Date": "01/09/2022", "Open": 7284.15000, "High": 7284.15000, "Low": 7131.69000, "Close": 7148.50000}, {"Symbol": "UKX:IND", "Date": "31/08/2022", "Open": 7361.63000, "High": 7378.44000, "Low": 7263.62000, "Close": 7284.15000}, {"Symbol": "UKX:IND", "Date": "30/08/2022", "Open": 7427.31000, "High": 7486.40000, "Low": 7351.12000, "Close": 7361.63000}, {"Symbol": "UKX:IND", "Date": "26/08/2022", "Open": 7479.74000, "High": 7530.65000, "Low": 7422.02000, "Close": 7427.31000}, {"Symbol": "UKX:IND", "Date": "25/08/2022", "Open": 7471.51000, "High": 7535.70000, "Low": 7469.17000, "Close": 7479.74000}, {"Symbol": "UKX:IND", "Date": "24/08/2022", "Open": 7488.11000, "High": 7488.12000, "Low": 7410.40000, "Close": 7471.51000}, {"Symbol": "UKX:IND", "Date": "23/08/2022", "Open": 7533.79000, "High": 7533.79000, "Low": 7467.56000, "Close": 7488.11000}, {"Symbol": "UKX:IND", "Date": "22/08/2022", "Open": 7550.37000, "High": 7550.41000, "Low": 7491.26000, "Close": 7533.79000}, {"Symbol": "UKX:IND", "Date": "19/08/2022", "Open": 7541.85000, "High": 7578.85000, "Low": 7513.26000, "Close": 7550.37000}, {"Symbol": "UKX:IND", "Date": "18/08/2022", "Open": 7515.75000, "High": 7541.89000, "Low": 7493.66000, "Close": 7541.85000}]
```

Which we see looks like a list of python dicts. And the actual summary data when downloaded will look like this:

	Symbol	Date	Open	High	Low	Close
0	UKX:IND	01/09/2022	7284.15	7284.15	7131.69	7148.50
1	UKX:IND	31/08/2022	7361.63	7378.44	7263.62	7284.15
2	UKX:IND	30/08/2022	7427.31	7486.40	7351.12	7361.63
3	UKX:IND	26/08/2022	7479.74	7530.65	7422.02	7427.31
4	UKX:IND	25/08/2022	7471.51	7535.70	7469.17	7479.74
...	...	...	...	...	...	...
6315	UKX:IND	18/09/1997	4994.50	5057.80	4993.60	5046.20
6316	UKX:IND	17/09/1997	5025.40	5035.30	5002.30	5013.10
6317	UKX:IND	16/09/1997	4892.80	4977.30	4876.80	4976.40
6318	UKX:IND	15/09/1997	4876.30	4902.90	4864.60	4902.90
6319	UKX:IND	12/09/1997	4851.20	4878.00	4833.90	4848.20
6320 rows x 6 columns						

After light visual inspection this appears to be fine. However, if there was an error with the data we would be thrown an error.

For example the initial attempt at importing the data frame threw this error:

`UnicodeDecodeError: 'utf-8' codec can't decode byte 0xff in position 0: invalid start byte`

And we quickly searched stackoverflow to find other users had corrected this by changing the encoding type to 'utf-16'. These are often the kinds of minor issues that we face.

We pip install a new library ( `mplfinance` ) specifically for finance charts and run the following code:

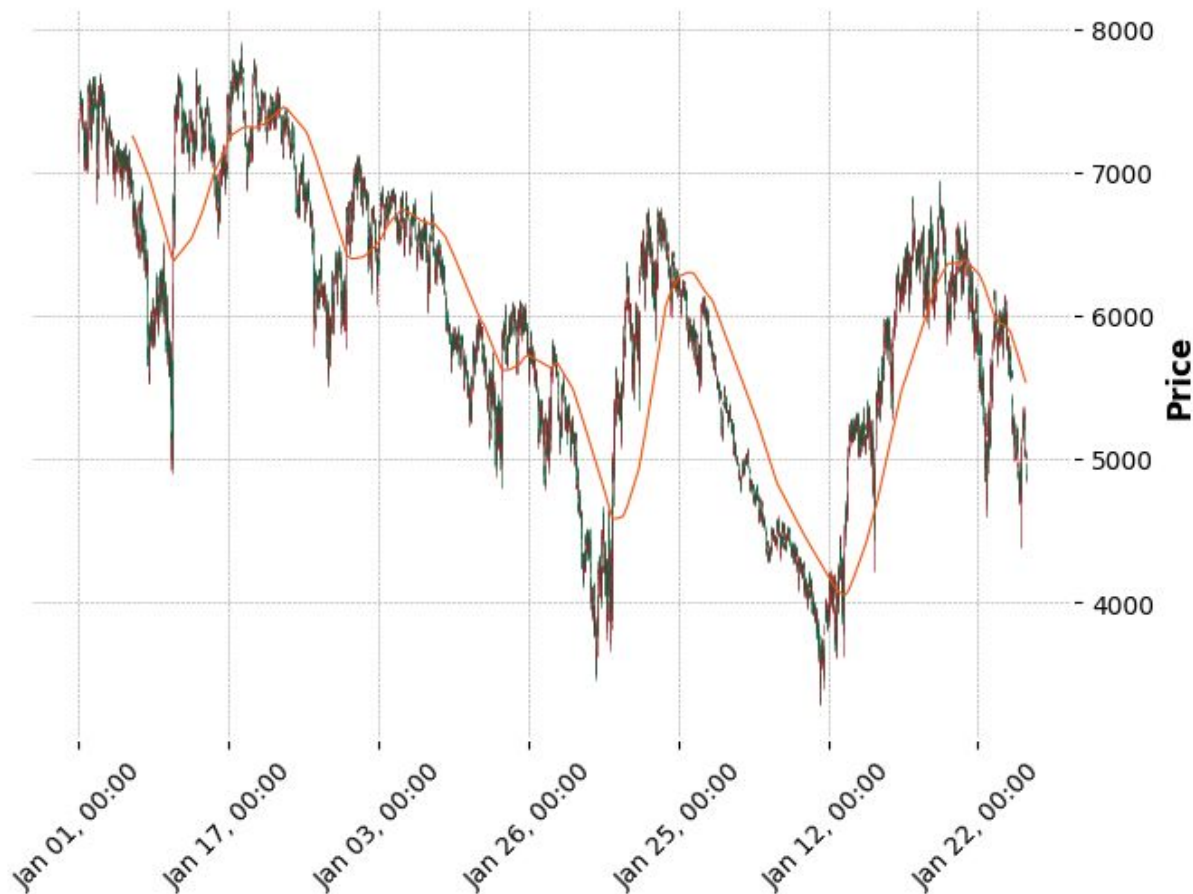
```
import mplfinance as mpf
import pandas as pd

df = pd.read_csv("C:/Users/admin/Downloads/markets_historical_ukx_ind.csv",
                 encoding = 'utf-16')

# format the data
df['Date']=pd.to_datetime(df['Date'],format='%d/%M/%Y').dt.date
df.index= pd.DatetimeIndex(df['Date'])
df = df.drop(['Date'], axis=1)

# https://github.com/matplotlib/mplfinance#usage
mpf.plot(df, type='candle', style='charles', mav=(360),
        warn_too_much_data=10000)
```

To get this:



Which is the open, high, low, close data with a 360 day moving average.

### [Cleaning data](#)

After we have gone to the extent of obtaining the data into a format that can be easily read and transferred into the code we then have the next task.

The least glamorous part of the task is cleaning and formatting the data. This is because the external world does not see the efforts of the data clean process which is often far greater than the output.

For example, in the download of the ftse 100 data, we had to get the original data into a format that is acceptable for the module to read (we could have opted for a different charting module) which needed to be like this:

	Symbol	Open	High	Low	Close
Date					
2022-01-01	UKX:IND	7284.15	7284.15	7131.69	7148.50
2022-01-31	UKX:IND	7361.63	7378.44	7263.62	7284.15
2022-01-30	UKX:IND	7427.31	7486.40	7351.12	7361.63
2022-01-26	UKX:IND	7479.74	7530.65	7422.02	7427.31
2022-01-25	UKX:IND	7471.51	7535.70	7469.17	7479.74
---	---	---	---	---	---
1997-01-18	UKX:IND	4994.50	5057.80	4993.60	5046.20
1997-01-17	UKX:IND	5025.40	5035.30	5002.30	5013.10
1997-01-16	UKX:IND	4892.80	4977.30	4876.80	4976.40
1997-01-15	UKX:IND	4876.30	4902.90	4864.60	4902.90
1997-01-12	UKX:IND	4851.20	4878.00	4833.90	4848.20
6320 rows x 5 columns					

Whilst the format looks very similar to the original data download, there were issues with:

- The dataframe index which needed to be a datetime format.
- The date column which was a series of strings to be converted to datetimes.
- The order of the columns.

Each of these was necessary to plot that particular candle chart and without any item the code would have thrown its respective error. And figuring each item out one-by-one was time consuming.

## Moving averages

The chart that we had offered a keyword argument `mav=()` which allows the user to plot a moving average or a series of moving averages.

```
import pandas as pd
from matplotlib import pyplot as plt

df = pd.read_csv("C:/Users/admin/Downloads/markets_historical_ukx_ind.csv",
                 encoding = 'utf-16')
df['Date']=pd.to_datetime(df['Date'], format='%d/%m/%Y')

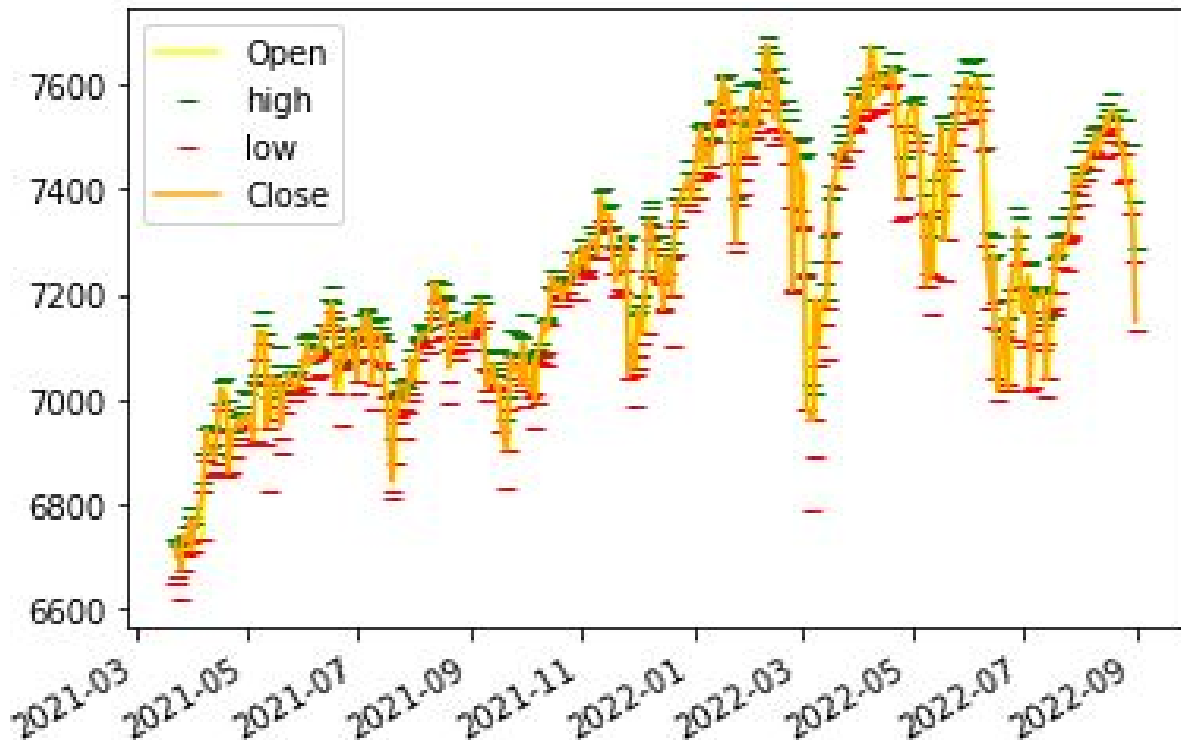
df = df.head(365)
df

fig, ax = plt.subplots()

ax.plot(df['Date'], df['Open'], '-', color='yellow', label='Open')
ax.plot(df['Date'], df['High'], '-', color='green', label='high')
ax.plot(df['Date'], df['Low'], '-', color='red', label='low')
ax.plot(df['Date'], df['Close'], '-', color='orange', label='Close')
fig.autofmt_xdate()
ax.legend()

plt.show()
```





However, we can improve this by adding new columns to the dataframe. Where for example, we add a moving average column and an exponential moving average column.

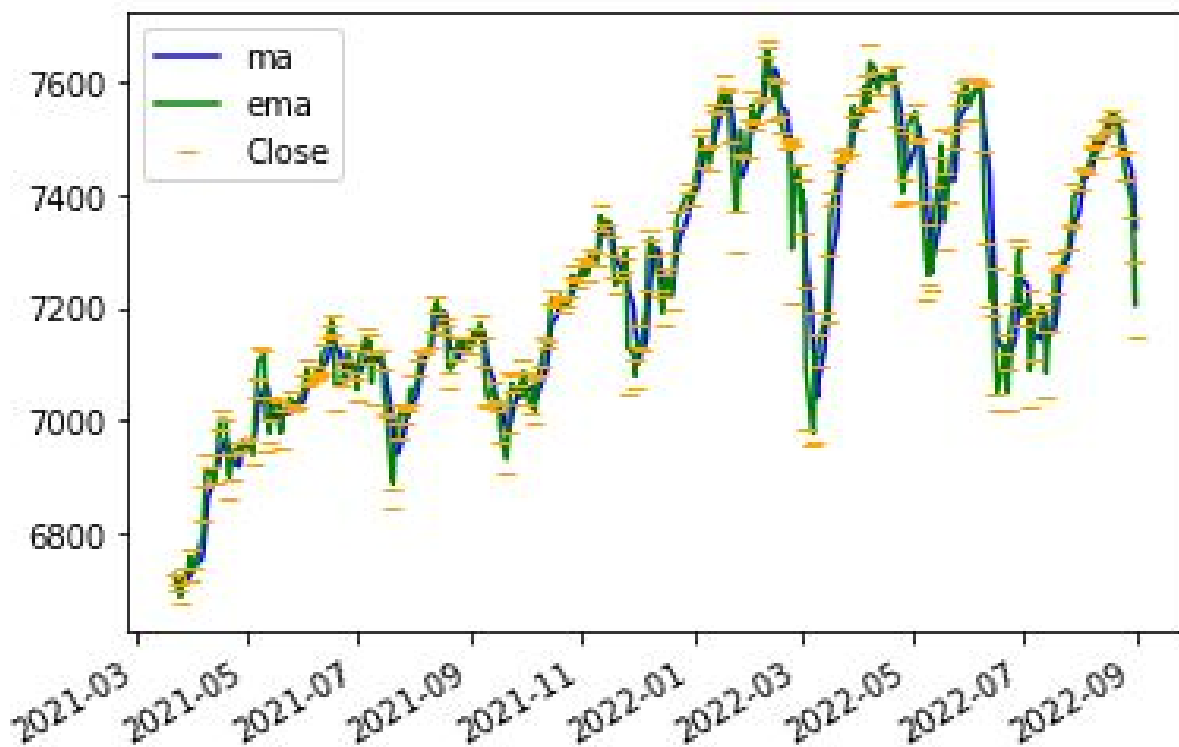
```
df['ma'] = df['Close'][::-1].rolling(window=5).mean()
df['ema'] = df['Close'][::-1].ewm(com=0.5).mean()

fig, ax = plt.subplots()

ax.plot(df['Date'], df['ma'], '-', color='blue', label='ma')
ax.plot(df['Date'], df['ema'], '-', color='green', label='ema')
ax.plot(df['Date'], df['Close'], '-', color='orange', label='Close')
fig.autofmt_xdate()
ax.legend()

plt.show()
```

Meaning that we can plot both against the daily close for example which looks like this:



But in fact, we could do any logical test on the data and flag this on the chart as a trading signal or point of interest.

The first step for completing such a task is to create a new column of interest in the dataframe. For example, let's say that we want to find drawdowns<sup>[14]</sup> in the market. Then we might set about looking for differences in rows as a percentage of the exponential moving average.

Then we could do this:

```
df['ma'] = df['Close'][::-1].rolling(window=5).mean()
df['ema'] = df['Close'][::-1].ewm(com=0.5).mean()
df['diff'] = df['Close'][::-1].diff(periods=5)
df['dd'] = df['diff'] / df['ema']

fig, ax = plt.subplots(2)

# ax[0].plot(df['Date'], df['ma'], '-', color='blue', label='ma')
ax[0].plot(df['Date'], df['ema'], '-', color='green', label='ema')
```

```

ax[0].plot(df['Date'], df['Close'], '-', color='orange', label='Close')
ax[0].legend()

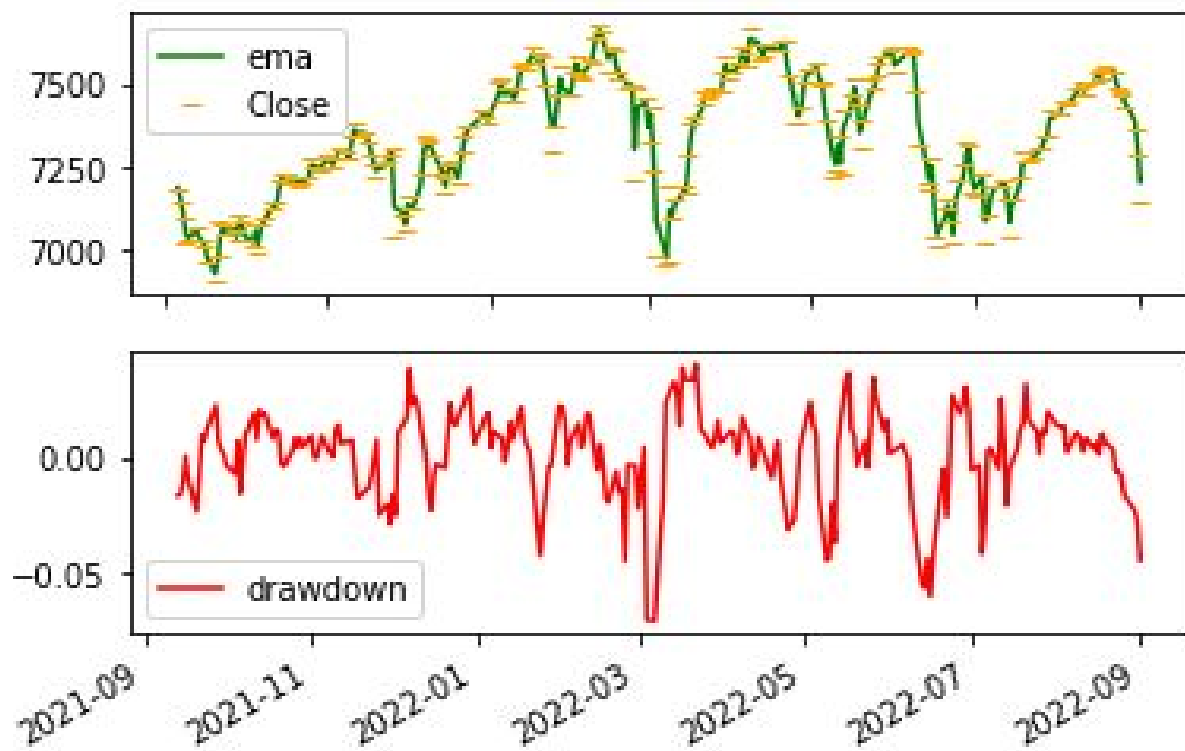
ax[1].plot(df['Date'], df['dd'], '-', color='red', label='drawdown')
ax[1].legend()

fig.autofmt_xdate()

plt.show()

```

Which produces this:



And then we might set about flagging those values into the system by seeking those that have passed some kind of threshold level, like a moving standard deviation or an absolute value.

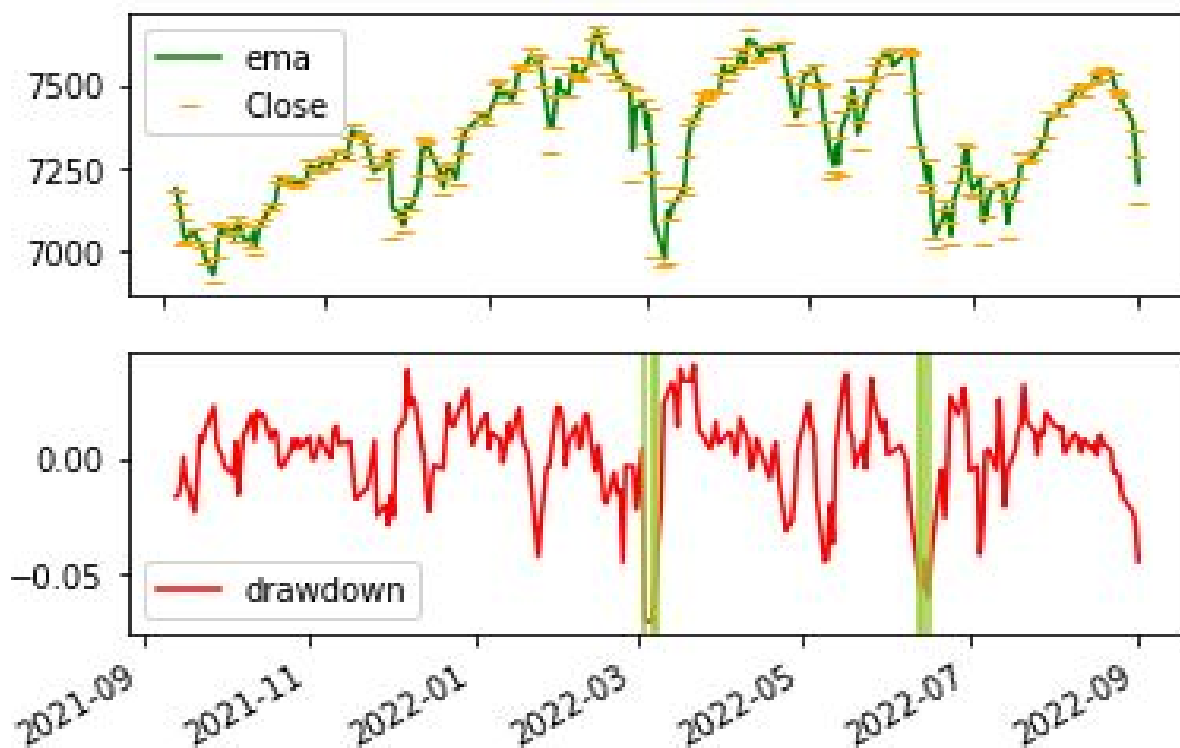
```

# find dates of interest
dd_dates = df[df['dd'] < -0.05]['Date']
for dd in dd_dates:

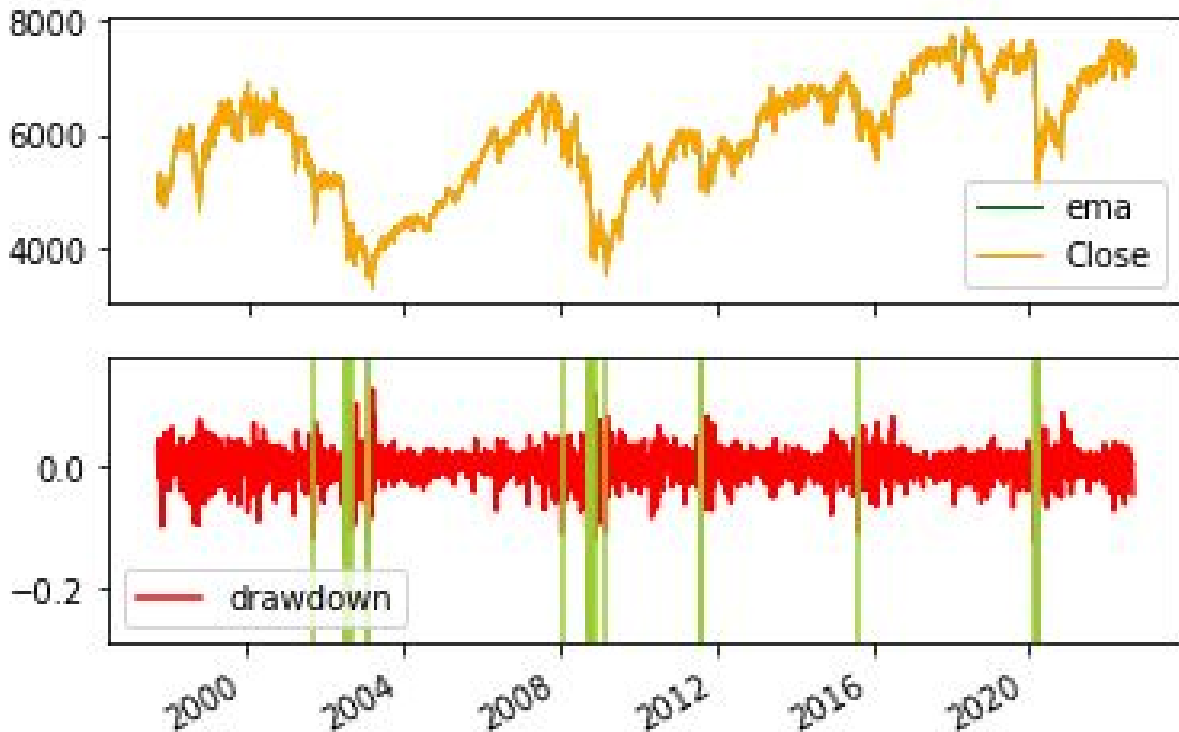
```

```
ax[1].axvline(x=dd, color='yellowgreen')  
ax[1].legend()
```

We find and flag the dates where drawdowns are more than 5% over any 5 day period. Which return this visually and can be passed into the system as a list or text or json file as per the user requirement.



Or allowing for all 6250 rows data we find all periods where the drawdown was  $> 10\%$  over a 10 day period like this:



In summary, we now have a very powerful tool at our fingertips. We can inspect *historical data* and we could also do the same in *real-time*. We can look for and flag trading signals and even automate the initiation of trades or other processes based on the information.

Whilst the techniques shown have fallen into the trading section, the implications are far reaching as we can use similar methods in all industry, science and engineering.

For example, a response procedure for self-driving cars purely based on signal inputs collected by a camera ccd array.

## Reading data

We have touched on getting data into our code with the `open` file method and the `read_csv` function in the pandas module.

However, it is quite common to need to read excel files and also json files which is favoured for human readability and nesting features.

Info	JSON	CSV
File size	Large	Compact
Data type	Uses Javascript data types.	Does not use a data type.
Hierarchy	Supports hierarchical and relational data	Flat files
Uses	Stores and exchanges the syntax of data in arrays, objects, etc.	Stores tabular data in a delimited text file.

Then there are also xml and html tables from the web and then there are queries from a whole host of databases that are sql or no-sql.

Each source comes with its own nuanced challenge and the final goal is to get the data into a python object. From there manipulations can be done.

We can see how JSON differs from CSV in the sense that it permits for the hierarchy.

```
{
  "index":
    {"ftse100":{"members":100,"country":"uk"},
     "sp500":{"members":500,"country":"usa"},
     "dax":{"members":40,"country":"germany"},
     "cac":{"members":30,"country":"france"}
    },
  "currency":["gbp","usd","eur"],
  "transaction fee":[
    {"europe":100},
    {"asia":50},
```

```
{
  "america": {
    "north": 100,
    "south": 25
  },
  "passcode": 123456
}
```

The nice part is that we can read the file in the same way that we read text files.

```
import json

with open('C:/test/test.json') as f:
    d = json.load(f)
```

And the variable `d` is type inferred as a dict and assigned the entire content of the json as the {key, value} pairs of the dict. So accessing the nested data is the same as accessing any part of a dict by traversing down the branches.

```
d['transaction fee']
# [{'europe': 100}, {'asia': 50}, {'america': {'north': 100, 'south': 25}}]

d['transaction fee'][2]
# {'america': {'north': 100, 'south': 25}}

d['transaction fee'][2]['america']['south']
# 25

d['index']['sp500']['members']
# 500
```

Whilst it is possible to store the same type of data into a csv file, the format of the JSON for this task is far more logical.

The user can then edit the dict as they need or even create a new dict and write the data back in a very similar manner.

```
# edit the price of asia (existing key)
d['transaction fee'][1]['asia'] = 75
```

```

# edit number of members in cac (existing key)
d['index']['cac']['members'] = 40
d['index']['dax']['members'] = 30

# add a new index (new key:value pair)
d['index']['ta'] = {'members':125,'country':'israel'}

# write to file
with open('C:/test/output.json', 'w') as f:
    json.dump(d, f, indent=4)

```

Produces a new JSON file that looks like this:

```

{
  "index": {
    "fise100": {
      "members": 100,
      "country": "uk"
    },
    "sp500": {
      "members": 500,
      "country": "usa"
    },
    "dax": {
      "members": 30,
      "country": "germany"
    },
    "cac": {
      "members": 40,
      "country": "france"
    },
    "ta": {
      "members": 125,
      "country": "israel"
    }
  },
  "currency": [

```



```
{
  "currency": [
    "gbp",
    "usd",
    "eur"
  ],
  "transaction fee": [
    {
      "europe": 100
    },
    {
      "asia": 75
    },
    {
      "america": {
        "north": 100,
        "south": 25
      }
    }
  ],
  "passcode": 123456
}
```

Whilst the file is larger in size, we can clearly see the advantage and convenience of the JSON data format with python.

## Cosine similarity

This problem tends to manifest itself in the Natural Language Processing (NLP) field. In particular Cosine similarity is a metric used to measure how similar the documents are irrespective of their size. Mathematically, it measures the cosine of the angle between two vectors projected in a multi-dimensional space.

However, it has a useful case in finance too and in particular in indices. Take for example two bond indices which contain, say 1000 bonds in each. How could we measure the similarity between these two indices?

Well, this is identical to comparing two lists of strings to each other. So if we can come up with a technique, then we can state the “similarity”.

Basic examples are:

- [a,b,c] and [a,b,c] are 100% similar
- [a,b,c] and [d,e,f] are 0% similar
- [a,b,c] and [a,j,k] are 33% similar

But what about:

- [a,b,c] and [c,a,b] which contain the same letters but in a different order.
- [a,b,c] and [f,g,a] have one matching letter but in a different position.

We can say that for indices, the *order does not matter*, which reduces the problem to the first three cases.

And what about lists of different sizes ? which pair has a greater similarity:

- [a,b,c] with [a,b,c,d]
- [a,b,c] with [a,b,c,d,e,f g]

Again, we might say that the first pair are more similar because more of the items that do exist reside in both lists compared to the second pair.

We attempt to take a mathematical approach to this and in particular in python make use of dictionaries and lists or we could use the `cosine_similarity` method from the `sklearn` module.

So we now set out a recipe for how to do this:

1. Find all the unique items in both lists.
2. Compile two side by side vectors (of same length).
3. Perform a cosine measure (dot product).

Let's create a dict containing two example indices and operations to today the data up such that we can display a dataframe.

```
import pandas as pd

d = {'first_index_names':['d','c','b','a'],
```

```

    'first_index_weights':[0.2, 0.2,0.3,0.3],
    'second_index_names':['a', 'j', 'k', 'd', 'e'],
    'second_index_weights':[0.1, 0.1, 0.1,0.1,0.6]}

# combine names and weights
d['first_index']=zip(d['first_index_names'],d['first_index_weights'])
d['second_index']=zip(d['second_index_names'],d['second_index_weights'])

# check that the weights sum to 1
if sum(d['first_index_weights'])==1: pass
else: print('normalise first index values')
if sum(d['second_index_weights'])==1: pass
else: print('normalise second index values')

# create a unique index list
d['unique_names'] = d['first_index_names'] + d['second_index_names']
d['unique_names'] = list(set(d['unique_names']))
d['unique_names'].sort()

# create nulls such that the lists are of the same length
for i in d['unique_names']:
    if i not in d['first_index_names']:
        d['first_index_names'].append(i)
        d['first_index_weights'].append(0)
    if i not in d['second_index_names']:
        d['second_index_names'].append(i)
        d['second_index_weights'].append(0)

df = pd.DataFrame(d)

# order weights to unique list
dff['first_index_weights_sorted']=dff['first_index'].apply(lambda x: x[1])
dff['second_index_weights_sorted']=dff['second_index'].apply(lambda x: x[1])

# keep relevant columns
df = df.filter(['unique_names','first_index_weights_sorted',
               'second_index_weights_sorted'], axis=1)

```

```
df = df.rename(columns={"first_index_weights_sorted": "first_index",
                        "second_index_weights_sorted": "second_index"})
df
```

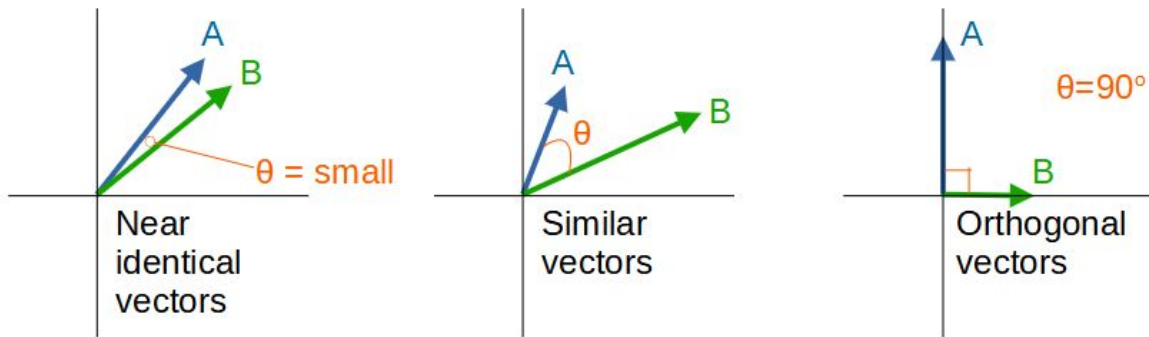
This is what we get as a result:

	unique_names	first_index	second_index
0	a	0.2	0.1
1	b	0.2	0.1
2	c	0.3	0.1
3	d	0.3	0.1
4	e	0.0	0.6
5	j	0.0	0.0
6	k	0.0	0.0

We have two side by side indices both on the same length and in the same order. Any items that don't contain values are given the value zero so there are no empty spaces. The task for computing a similarity has become immensely clearer.

However the data came in, be that via a csv file or a json or via list construction it is always desirable to reshape the lists to look like the above.

We now consider the mathematics of the Cosine similarity by thinking about the geometry of vectors.



In particular, two vectors are identical where the cosine of the angle between them is one and also the magnitude of the vector (referred to as the vector Norm or its length) are the same.

The general formula for cosine similarity is this:

$$\text{similarity} = \cos(\theta) = \frac{\underline{\underline{a \cdot b}}}{|a||b|}$$

Which reduces to:

$$\text{similarity} = \frac{\underline{\underline{a \cdot b}}}{|a||b|} = \frac{\sum_{i=1}^n (a_i b_i)}{\sqrt{\sum_{i=1}^n a_i^2} \sqrt{\sum_{i=1}^n b_i^2}}$$

Because the lists are homogeneous we can produce this in a few lines of code.

```
import math
aibi = sum(df['first_weight'] * df['second_weight'])
a_norm = math.sqrt(sum(df['first_weight']**2))
b_norm = math.sqrt(sum(df['second_weight']**2))
similarity = aibi / (a_norm * b_norm)

print('the similarity is:', similarity)
# result is 0.155
```

We can now compare any two indices and we could test the code with logical inputs.

For example, the following two lists have no similarity.

```
d = {'first_index_names':['a','b','c'],
     'first_index_weights':[0.2, 0.2, 0.6],
     'second_index_names':['d', 'e', 'f'],
     'second_index_weights':[0.1, 0.1, 0.8]}
```

And these lists are 100% similar.

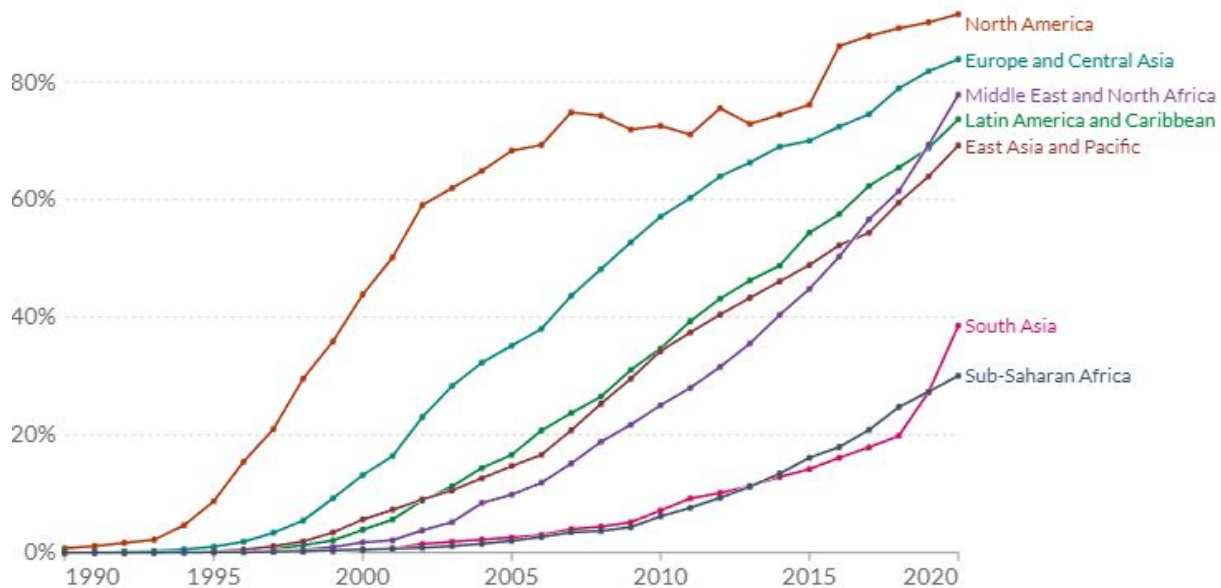
```
d = {'first_index_names':['a','b','c'],
     'first_index_weights':[0.2, 0.2, 0.6],
     'second_index_names':['c', 'a', 'b'],
     'second_index_weights':[0.6, 0.2, 0.2]}
```

Which works nicely. And everything in between will be between 0% and 100%.

This is a very useful tool as it becomes possible to predict and explain the performance of two portfolios or indices prior to performing a historical price comparison as indices with a *high degree of similarity* are expected to be *correlated in price*.

## The web

The web provides a vast assortment of data and a myriad of use cases that spawns nearly all the information that we are aware of. The Global digital population as of April 2022 was over 5 billion internet users and nearly 5 billion social media users, but the reality is that in certain parts of the world, the share of the population using the internet is much closer to 100%.



We can automate the extraction of this resource with nearly any programming language and python has modules that make this relatively easy.

We will explore two types of connection. The HTTP request and WebSockets.

## HTTP request

The main module for http requests is aptly named `requests`. The `requests` module allows you to send HTTP requests using Python. The HTTP request returns a `Response` Object with all the response data (content, encoding, status, etc).

There main functions are:

- `get`: for getting data
- `post`: for sending data

The primary argument requires the url<sup>[15]</sup> and then there are optional arguments that can be included, for example `auth` for authentication and requires a ('user', 'pass') tuple or `timeout` to guarantee that the connection will be timed out.

The user can send a get request to any website and will receive a response.

```
import requests

r = requests.get('https://w3schools.com')
print(r.status_code)
```

The response comes with a status code.

There are many status codes categorised as follows:

- Informational responses (100–199)
- Successful responses (200–299)
- Redirection messages (300–399)
- Client error responses (400–499)
- Server error responses (500–599)

The main response is: 200. This means the request was successful. Other common ones are:

- 400: bad request - server cannot process the request
- 401 Unauthorised - needed a password/authenticate
- 403 Forbidden - The client does not have access rights to the content

We can check requests in a `try-except` block:

```
import requests

try:
    # a site that does NOT exist
    r = requests.get('https://blahblah.com')
    print(r.status_code)
except Exception as e:
    print('failed to connect')
```



```

print(e)

try:
    # a site that exists
    r = requests.get('https://google.com')
    print(r.status_code)
except Exception as e:
    print('failed to connect')
    print(e)

```

but given that we know that any given site is valid it is more common to just look at the response `status_code` like this:

```

if r.status_code==200:
    print('proceed with code...')
else:
    print('can not proceed:', r.status_code)

```

Aside from the status code, we can read the content of the server's response. This typically comes in 4 types:

1. Text. `r.text`
2. Binary. `r.content`
3. JSON. `r.json()`
4. Raw (less common). `r.raw`

Generally, the requests module makes educated guesses about the encoding of the response based on the HTTP headers.

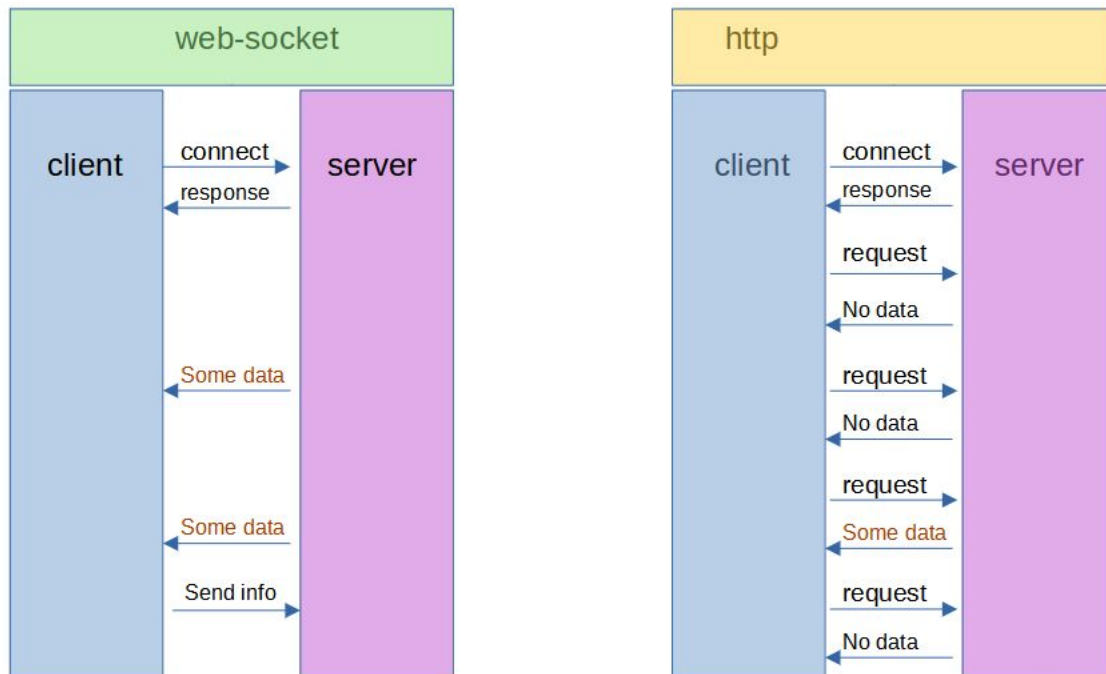
However, the user can find the encoding by doing `r.encoding` .

## Websockets

WebSocket is an event-driven protocol, which means you can actually use it for truly real time communication. Unlike HTTP, where you have to

constantly request updates, with websockets, updates are sent immediately when they are available.

Take for example a financial exchange which has continually updating prices for different securities throughout the day. The websocket allows the user to log in once to the exchange and then listen to the prices for various securities as and when they happen compared to the HTTP version where the user would continually have to connect and make requests for prices.



The has the advantage that it pushes messages between the **server** and the **client** *on demand* as they need them.

So let's look at the basic setup for both sides:

- Server
- Client

Here's a WebSocket server example.

```
import asyncio
import websockets

async def hello(websocket, path):
```

```

name = await websocket.recv()
print(f"< {name}")

greeting = f"Hello {name}!"

await websocket.send(greeting)
print(f"> {greeting}")

start_server = websockets.serve(hello, "localhost", 8765)

asyncio.get_event_loop().run_until_complete(start_server)
asyncio.get_event_loop().run_forever()

```

And here is the corresponding client example.

```

import asyncio
import websockets

async def hello():
    uri = "ws://localhost:8765"
    async with websockets.connect(uri) as websocket:
        name = input("What's your name? ")

        await websocket.send(name)
        print(f"> {name}")

        greeting = await websocket.recv()
        print(f"< {greeting}")

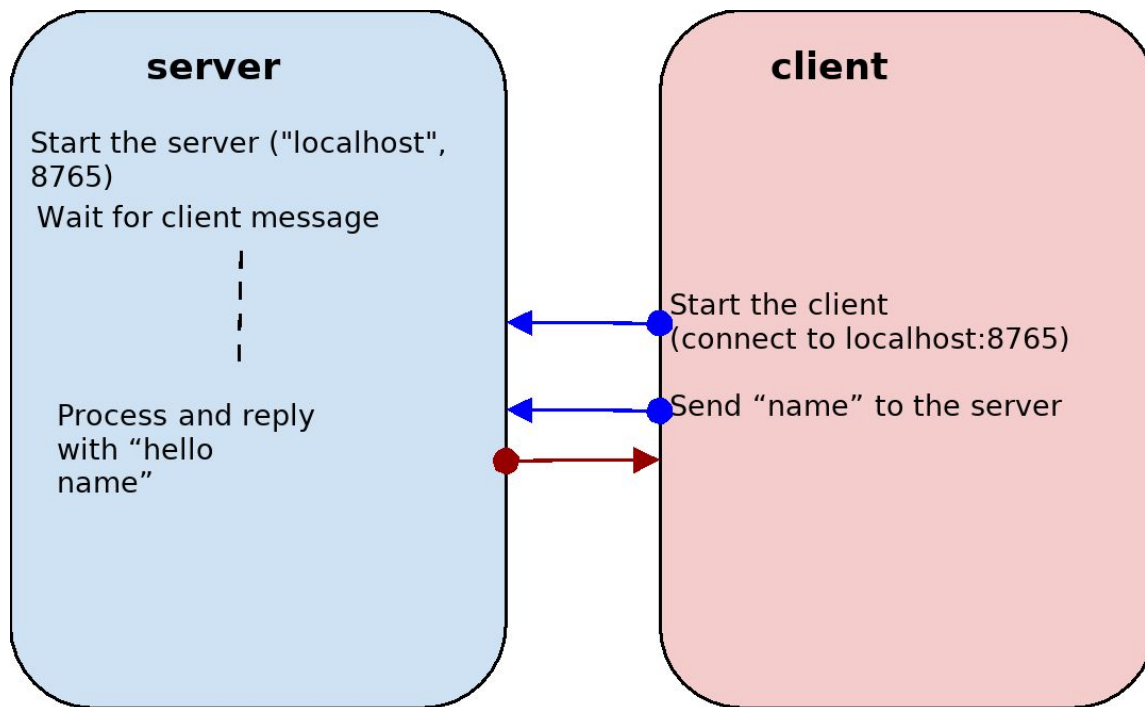
asyncio.get_event_loop().run_until_complete(hello())

```

And this was all that we needed to have a server-client connection.

In the example given, both pieces of code are run at the same time. **The server** sits and waits for the connection to be made. **The client** connects to the server (on localhost:8765 in the example) and sends the string name. If **the server** receives a message, then it sends a reply back. And finally, **the client** receives the reply.

So the example did this:



Whilst the example is basic, it contains all that is necessary for both the client and the server to talk to each other. Processes could be done at either end and the results sent back to the other one.

This means that we could, for example, send small bits of information from the client to the server and have the server compute a heavy process (like a heavy duty cloud server) or alternatively the other way around if we chose to.

So we could have our functions and modules live on the server. This is, in fact, the basis for the internet and how our web browsers and mobile devices interact.

Here is some code where the server sends messages which are accessed by a browser. In the case of the example it is just sending the current time every 3 seconds, but similar methods could be used to stream stock prices, news headlines, security data and any other data that is dynamic.

```
import asyncio
```

```

import datetime
import random
import websockets

async def time(websocket):
    while True:
        now = datetime.datetime.utcnow().isoformat() + "Z"
        await websocket.send(now)
        await asyncio.sleep(random.random() * 3)

start_server = websockets.serve(time, "127.0.0.1", 5678)

asyncio.get_event_loop().run_until_complete(start_server)
asyncio.get_event_loop().run_forever()

```

And this is the HTML page that reads the same data and will open in any standard browser.

```

<!DOCTYPE html>
<html>
  <head>
    <title>WebSocket demo</title>
  </head>
  <body>
    <script>
      var ws = new WebSocket("ws://127.0.0.1:5678/"),
          messages = document.createElement('ul');
      ws.onmessage = function (event) {
        var messages = document.getElementsByTagName('ul')[0],
            message = document.createElement('li'),
            content = document.createTextNode(event.data);
        message.appendChild(content);
        messages.appendChild(message);
      };
      document.body.appendChild(messages);
    </script>
  </body>
</html>

```

You will notice that the body of the code contains a script tag which contains javascript which as we are now aware is similar to python in many ways but gets its syntax of curly braces and semicolons from the c family of languages.

With this HTML containing javascript, we are able to see the time stream down the screen every 3 seconds.

Generally speaking, a mobile device, which is low powered, is the **client**. On the other hand, the web host, which has a lot of power, is the **server**. So our low powered device sends lightweight messages (instructions) to the hosts server.

And large processes which consume a lot of energy can occur at the server end. These are then sent back to the mobile device, which means that the small batteries in mobile phones are sufficient.

## Asynchronous

In the websockets chapter we came across a function that we have not seen before. The usual format for a function is as follows.

```
def normal_function(a, b):  
    """ this is a normal function """  
    c = a + b  
    return c
```

But we now see a function definition preceded by the keyword `async` and we also see the `await` keyword.

```
async def normal_function(a, b):  
    """ this is a normal function """  
    c = a + b  
    await asyncio.sleep(0)  
    return c
```

Sometimes we want to run routines at the same time. We do not want to wait for one to finish before starting the next. A real world example of an asynchronous process might be making a cup of tea. If we switch the kettle on first then we can prepare the tea cups with bags, milk and sugar whilst

the kettle is boiling. We were able to do two processes at the same time which saved time.

The `await` keyword is where the process can hand over control to another process and in the case of the tea example, it would logically occur once the kettle was switched on. We now have two processes running simultaneously [1] the kettle boiling, [2] the cups being prepared.

These routines that are run at the same time are termed coroutines and require some use of asynchronous input and output (Async IO) in Python. Async IO is related to multiprocessing and threading in the sense that languages need some kind of process that **does not block code execution** whilst other parts of the code run. And this is what python has to offer, so let's get a deeper understanding.

Async type	cpu	description
Parallelism (multiprocessing)	many	spreading tasks over a computer cpu (or cores)
Concurrency	one	multiple tasks have the ability to run in an overlapping manner (controlled by code).
Threading	one	multiple threads take turns executing tasks. One process can contain multiple threads.

A coroutine is a function that can suspend its execution before reaching return, and it can indirectly pass control to another coroutine for some time.

Before we get to this point we have to understand that this is a new technology (python 3.7 and above) which has been built from existing tech and we are only looking at the top level as a deep dive would require a book in itself. The components to understand are:

1. The async ecosystem
2. The event loop (all languages)
3. Using coroutines
4. How coroutines work
5. What comes included in the ecosystem
6. Typical web applications
7. Interacting with the blocking world
8. Error handling, testing, debugging

Each of the asynchronous methods has its advantages and disadvantages.

**With multiprocessing**, the user is able to use multiple processors at any point in time, but **the con is that processors are limited**. **With threading** the user has an easy way of sharing processes on the same thread but the con is that the **computer takes control of when this happens** which can give rise to race<sup>[16]</sup> conditions. **With asyncio** the user can achieve the same as threading and also keep control, but the con is that **the functions need to be defined as async** and there are additional keywords and terminology to use.

Here is a classic example of race conditions in threading which gives rise to additional implementations being required and ultimately asyncio.

```
import threading
import time

x = 10

def increment(by):
    global x

    local_counter = x
    local_counter += by
```



```

time.sleep(1)

x = local_counter
print(f'{threading.current_thread().name} inc x {by}, x: {x}')

def main():
    # creating threads
    t1 = threading.Thread(target=increment, args=(5,))
    t2 = threading.Thread(target=increment, args=(10,))

    # starting the threads
    t1.start()
    t2.start()

    # waiting for the threads to complete
    t1.join()
    t2.join()

    print(f'The final value of x is {x}')

for i in range(10):
    main()

```

The results look like this:

```

Thread-56 (increment) inc x 10, x: 20Thread-55 (increment) inc x 5, x: 15

The final value of x is 15
Thread-57 (increment) inc x 5, x: 20Thread-58 (increment) inc x 10, x: 25

The final value of x is 25
Thread-60 (increment) inc x 10, x: 35Thread-59 (increment) inc x 5, x: 30

The final value of x is 30

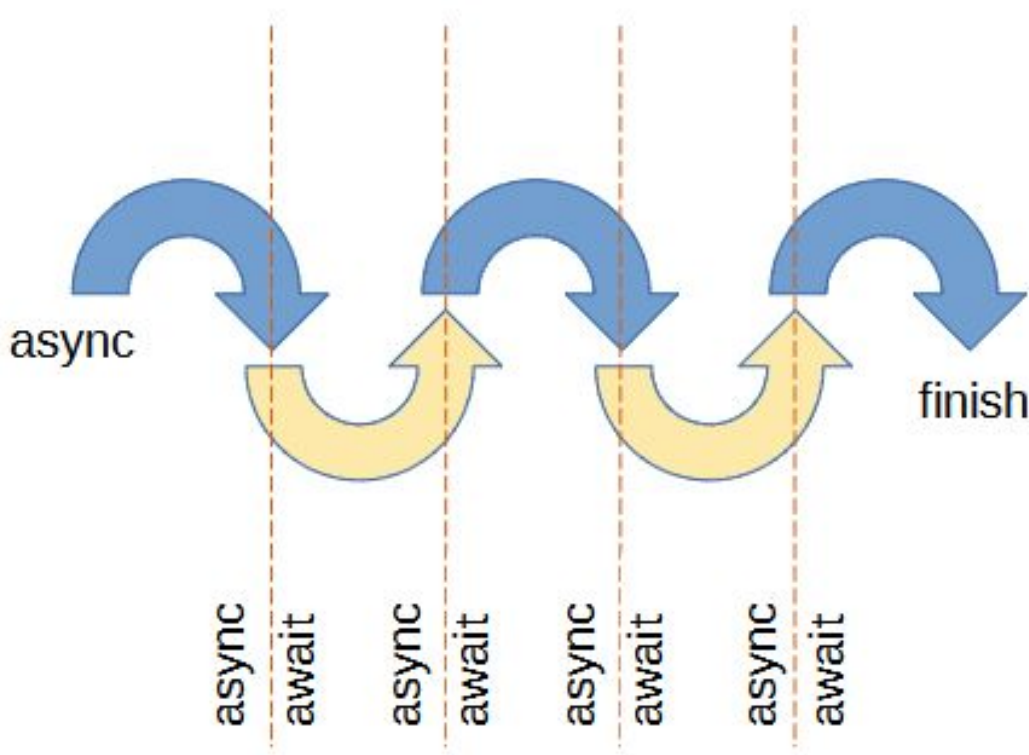
```

```
Thread-61 (increment) inc x 5, x: 35
Thread-62 (increment) inc x 10, x: 40
The final value of x is 40
Thread-64 (increment) inc x 10, x: 50Thread-63 (increment) inc x 5, x: 45
The final value of x is 45
```

Which is inconsistent at each loop and this was caused by the processor allocating control in an “unexpected” way.

There are ways to address this in threading with the GIL<sup>[17]</sup> and even better, using the `await` keyword in an `async` function for proper handover of the function and control of the program.

The important thing to note about asynchronous input and output routines is that the control of the program flow can now be definitely broken down into linear cycles where a callback function is only invoked once the preceding function allows for this.



The code, whilst asynchronous, is also systematic and definite in its objectives.

## Graphical user interfaces

The web is by far the go to place and python has its own offerings in the form of the Django and Flask web frameworks.

Aside from the web, there are bespoke graphical user interfaces (GUI's) that allow a user to build local apps on their devices which are predominantly desktop computers, laptops, tablets and mobile devices.

At the time of writing, at a glance, there were roughly 30 GUI's which operate across a variety of operating systems. Each will have its own specific features and advantages.

However, we will focus on one particular GUI called `tkinter`<sup>[18]</sup>. The main reasons that we focus on `tkinter` are:

1. It is fast.
2. It comes bundled with Python.
3. The general concepts of using GUI's are broadly the same. And we will justify why this general statement is true.

## Hello world GUI

Let's take a look at a basic hello world implementation using `tkinter`.

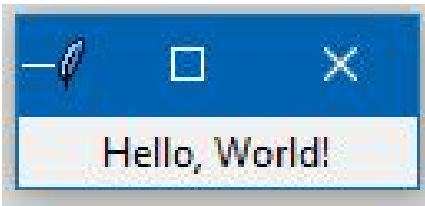
```
import tkinter as tk

# create an instance
root = tk.Tk()

# place a label on the root window
message = tk.Label(root, text="Hello, World!")
message.pack()

# keeps the window displaying and running until you close it
root.mainloop()
```

Which opens the GUI with the text hello world like this:



So out of the box, we were able to get a result quickly. All we had to do was this:

1. Import the module
2. Create an instance
3. Create a label with the “hello world” text
4. Pack the label
5. Run the main loop

Typically, we place the call to the `mainloop()` method as the last statement in a Tkinter program.

The components that are added are called “widgets”. We create widgets of our choice and then `pack` them to the container. If we forget to `pack` the widget, then it will be created but remain invisible. There are roughly 20 basic widgets to choose from, like scroll bars, checkboxes and entry boxes etc...

The reason why we said that the general concepts of all GUI’s are the same is because they tend to follow the same 5 step process as above.

What actually happens with all the drawing packages is that the screen refreshes itself many times per second (which is usually called the Frames Per Second, FPS) with the images being created or drawn sequentially one-by-one. This is more obvious for games modules like `pygame` where we might need to be careful to draw the background before the items.

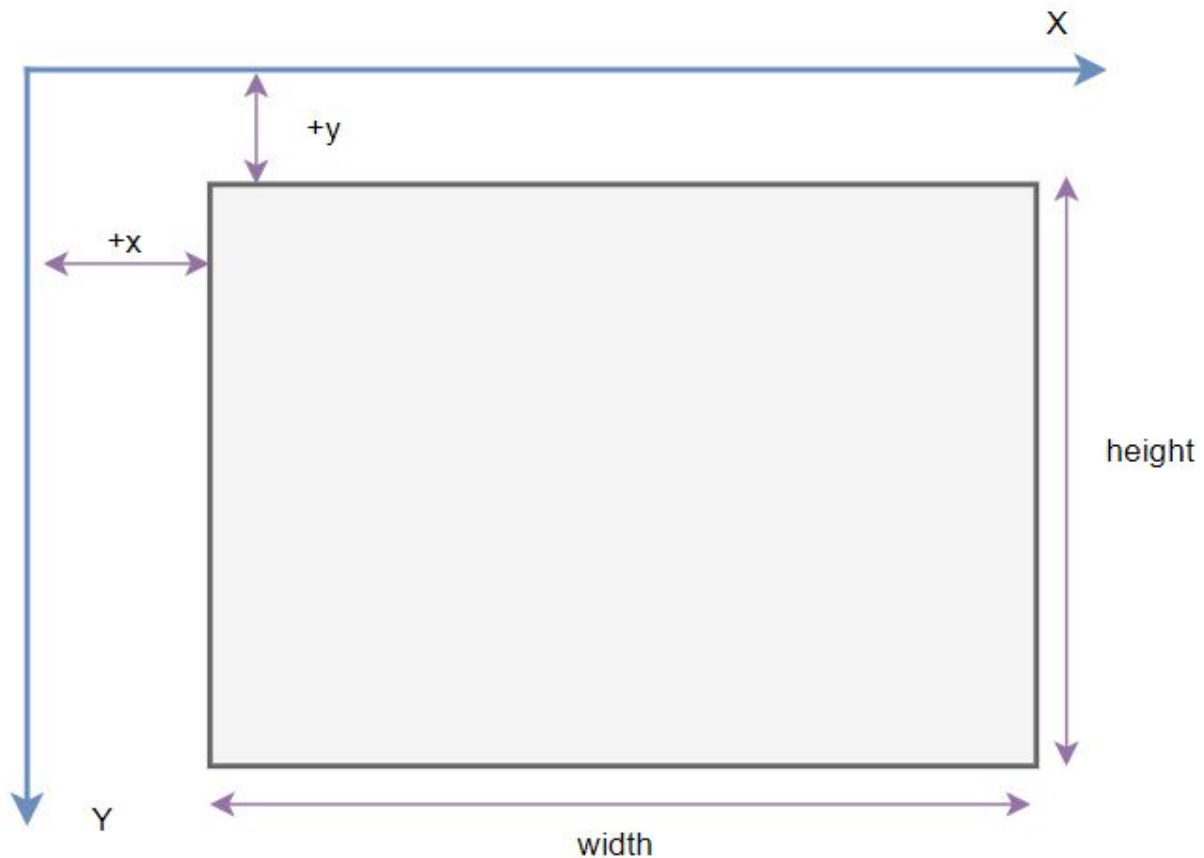
The hello world was great, but let's now extend upon this. We can give the GUI a title and a position and size on the screen:

```
root.title('Tkinter Window Demo')
root.geometry('600x400+200+50')
```

The geometry is made of two components:

- width \* height

- Position of top left corner relative to the screen



There are more options, but this is sufficient. In fact, omitting the top left position would leave a suitable default.

If we wanted to position and size the GUI relative to the screen, then we would need to get the screen dimensions and calibrate.

```
# get the screen dimension
screen_width = root.winfo_screenwidth()
screen_height = root.winfo_screenheight()

# the GUI width and height
window_width = int(screen_width / 3)
window_height = int(screen_height / 3)
```

We can do other things like:

- Control window size: `resizable()` , `minsize()` , `maxsize()`
- Adjust transparency: `attributes('-alpha',0.5)`
- Stacking order: `attributes('-topmost', 1)`
- Default icon: `iconbitmap('my_image.ico')`

We can implement all of the above.

```
import tkinter as tk

# create an instance
root = tk.Tk()
root.title('Tkinter Window Demo')

# get the screen dimension
screen_width = root.winfo_screenwidth()
screen_height = root.winfo_screenheight()

# the GUI width and height
window_width = int(screen_width / 3)
window_height = int(screen_height / 3)

# the GUI position
window_x_pos = int(screen_width * 0.1)
window_y_pos = int(screen_height * 0.1)

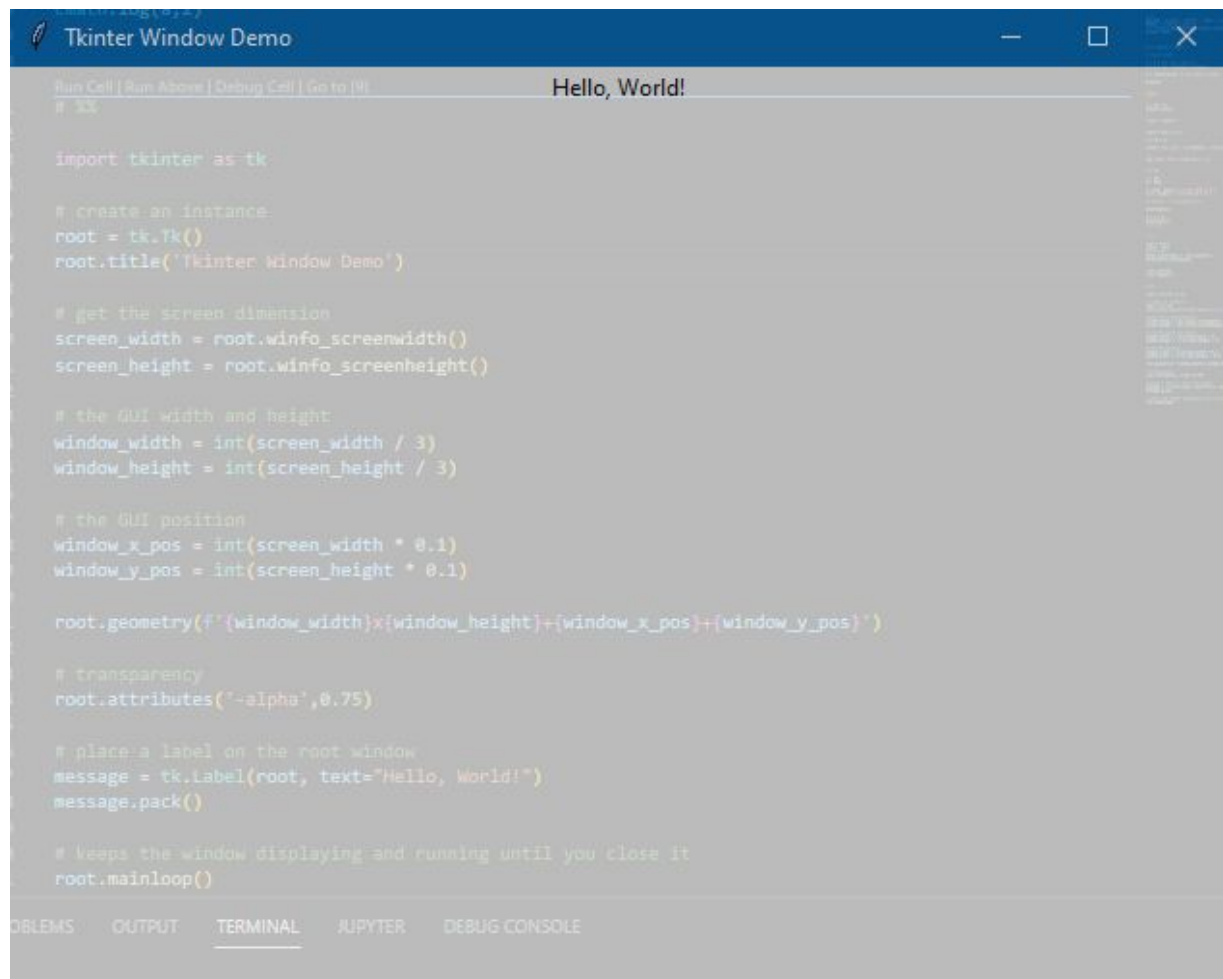
root.geometry(f'{window_width}x{window_height}+{window_x_pos}+{window_y_pos}')

# transparency
root.attributes('-alpha',0.75)

# place a label on the root window
message = tk.Label(root, text="Hello, World!")
message.pack()

# keeps the window displaying and running until you close it
root.mainloop()
```

Which gives this 25% transparent window which is positioned 10% from the top left hand corner of the screen and is  $\frac{1}{3}$  of the screen size in the x and y directions:



```
Run Cell | Run Above | Debug Cell | Go to | 91
# 33

Tkinter Window Demo
Hello, World!

import tkinter as tk

# create an instance
root = tk.Tk()
root.title('Tkinter Window Demo')

# get the screen dimension
screen_width = root.winfo_screenwidth()
screen_height = root.winfo_screenheight()

# the GUI width and height
window_width = int(screen_width / 3)
window_height = int(screen_height / 3)

# the GUI position
window_x_pos = int(screen_width * 0.1)
window_y_pos = int(screen_height * 0.1)

root.geometry(f'{window_width}x{window_height}+{window_x_pos}+{window_y_pos}')

# transparency
root.attributes('-alpha',0.75)

# place a label on the root window
message = tk.Label(root, text="Hello, World!")
message.pack()

# keeps the window displaying and running until you close it
root.mainloop()
```

So we have a GUI which we can customise reasonably well in terms of position and size etc.

So next, we can add some widgets.

Common widgets are buttons and entry boxes although there are many others. These will be sufficient for creating an example of receiving data and performing a basic operation.

We will create the button and entry box in a similar way that we created the label. The difference being that the button executes a function, which is usually referred to as a callback function. Callback<sup>[19]</sup> functions are very



common in modern programming and we have already seen them in Asynchronous programming.

Here is the button example, with a callback:

```
# make a button with a callback
def callback():
    """ this is called when the button is pressed """
    x = message.cget('text') # get current text
    if x == 'good night': message.config(text= 'hello world')
    else: message.config(text='good night')

button = tk.Button(root, text='press me', command=lambda: callback())
button.pack()
```

The button that we have created calls the callback function, which runs a little routine to toggle the text in the text box from “hello world” to “good night”.

It could be used for any other much more sophisticated routine in the same way as all other python functions. Also, we just used the name `callback()` for the purpose of illustration, but we could give a more appropriate name, like `change_message_text()` .

With the button sorted, we now move on to the entry widget, which allows the user to enter a single line of text.

```
# make an entry box
entry = tk.Entry(root)
entry.pack()
```

This is all that is needed and we can get the text from the box by using the `.get()` statement.

```
my_text = entry.get()
print(my_text)
```

And print the result straight to the console. We could add this into the callback function, so now the function does two things. [1] change the text label message, [2] print to the console.

Our basic code looks like this:

```
import tkinter as tk
```

```
# create an instance
root = tk.Tk()
root.title('Tkinter Window Demo')

# get the screen dimension
screen_width = root.winfo_screenwidth()
screen_height = root.winfo_screenheight()

# the GUI width and height
window_width = int(screen_width / 6)
window_height = int(screen_height / 3)

# the GUI position
window_x_pos = int(screen_width * 0.3)
window_y_pos = int(screen_height * 0.3)

root.geometry(f'{window_width}x{window_height}+{window_x_pos}+{window_y_pos}')

# place a label on the root window
message = tk.Label(root, text='hello world')
message.pack()

# make a button with a callback
def callback():
    """ toggle between 'good night' and 'hello world' """
    x = message.cget('text')
    my_text = entry.get()
    print(my_text)
    if x == 'good night': message.config(text='hello world')
    else: message.config(text='good night')

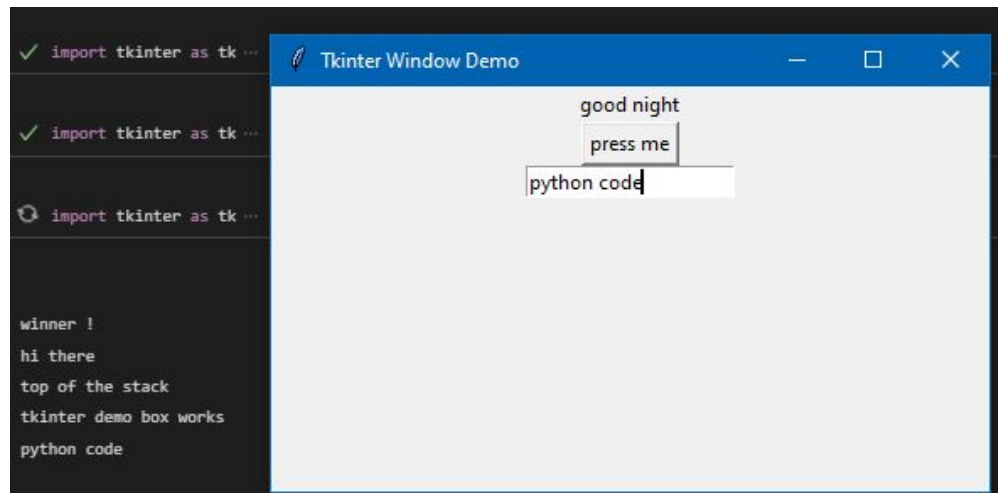
button = tk.Button(root, text='press me', command=lambda: callback())
button.pack()

# make an entry box
entry = tk.Entry(root)
```

```
entry.pack()

# keeps the window displaying and running until you close it
root.mainloop()
```

And the result looks like this:



There are many other things that we can do, for example, change the positions of the widgets in a grid instead of packing them one above the other.

## Password entry

Following on from the hello world example, which was nice, but just a starting point with examples. Here is a use case password sign in GUI which prompts the user for their email and password and then returns the result in a windows info box.

```
import tkinter as tk
from tkinter.messagebox import showinfo

# root window
root = tk.Tk()
root.geometry("300x150")
root.resizable(False, False)
root.title('Sign In')

# store email address and password
```

```
email = tk.StringVar()
password = tk.StringVar()

def login_clicked():
    """ callback when the login button clicked
    """
    msg = f'You entered email: {email.get()} and password: {password.get()}'
    showinfo(
        title='Information',
        message=msg
    )

# Sign in frame
signin = tk.Frame(root)
signin.pack(padx=10, pady=10, fill='x', expand=True)

# email
email_label = tk.Label(signin, text="Email Address:")
email_label.pack(fill='x', expand=True)

email_entry = tk.Entry(signin, textvariable=email)
email_entry.pack(fill='x', expand=True)
email_entry.focus()

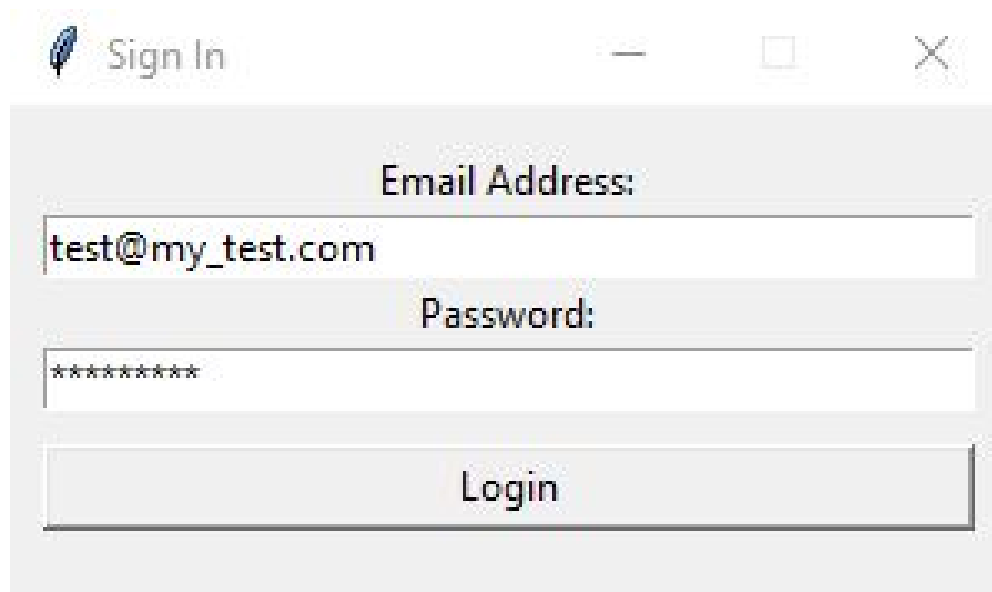
# password
password_label = tk.Label(signin, text="Password:")
password_label.pack(fill='x', expand=True)

password_entry = tk.Entry(signin, textvariable=password, show="*")
password_entry.pack(fill='x', expand=True)

# login button
login_button = tk.Button(signin, text="Login", command=login_clicked)
login_button.pack(fill='x', expand=True, pady=10)
```

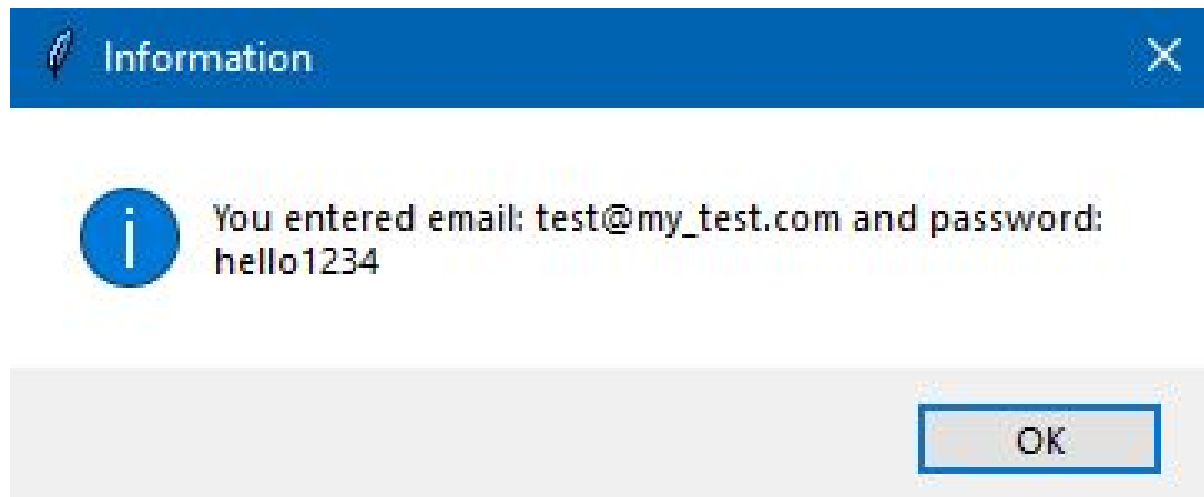
```
root.mainloop()
```

In just 20 lines of code, we have created a login box that can accept credentials and run a process. And it looks like this:



The image shows a graphical user interface window titled "Sign In". It has a standard window header with a feather icon, a title bar, and window control buttons (minimize, maximize, close). The main content area is light gray and contains three elements: a label "Email Address:" above a text input field containing "test@my\_test.com"; a label "Password:" above a text input field containing eight asterisks "\*\*\*\*\*"; and a large rectangular button labeled "Login" at the bottom.

And when the user clicks the “Login” button invokes another piece of code, via a callback. In the example, the code yields this:



The image shows an "Information" dialog box with a blue header bar containing a feather icon, the title "Information", and a close button (X). The main area is white and contains an information icon (blue circle with a white 'i') followed by the text "You entered email: test@my\_test.com and password: hello1234". At the bottom right, there is a button labeled "OK".

## Animations

All of the GUI features are nice. They give a user the ability to interact with the device via a clean and friendly interface via inputs and outputs

provisioned by the numerous widgets.

However, all of this has been rather static in the sense that our application sits and waits for a user action like a willing dog, but until that point is idle (although, we could remove the idleness by, say, having the device look at a share price and set an alert when it was breached).

We now move onto the dynamic part of coding with an example of two planets moving through space.

```
import tkinter as tk
import time

# by convention constants are in capitals
WIDTH = 500
HEIGHT = 300
velocity_x = 1
velocity_y = 1

win = tk.Tk()
win.title("moving planets")

canvas = tk.Canvas(win, width=WIDTH, height=HEIGHT)
canvas.pack()

photo_space = tk.PhotoImage(file="space.png")
photo_jup = tk.PhotoImage(file="jupiter.png")
photo_sat = tk.PhotoImage(file="saturn.png")

image_space = canvas.create_image(0,0,image=photo_space)
image_jup = canvas.create_image(0,0,image=photo_jup)
image_sat = canvas.create_image(450,0,image=photo_sat)

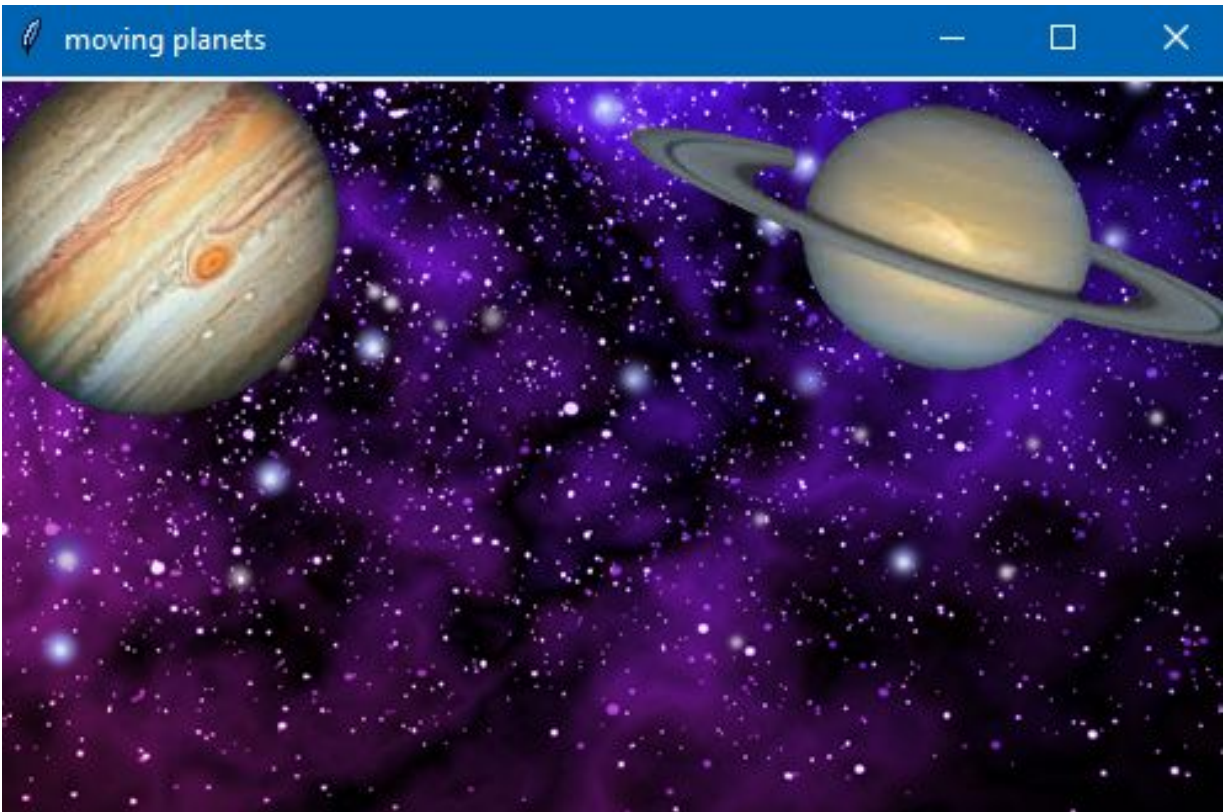
t0 = time.time()
while True:
    # plot positions to console
    coords_jup = canvas.coords(image_jup)
    coords_sat = canvas.coords(image_sat)
```

```
print('jup, sat:', coords_jup, coords_sat)

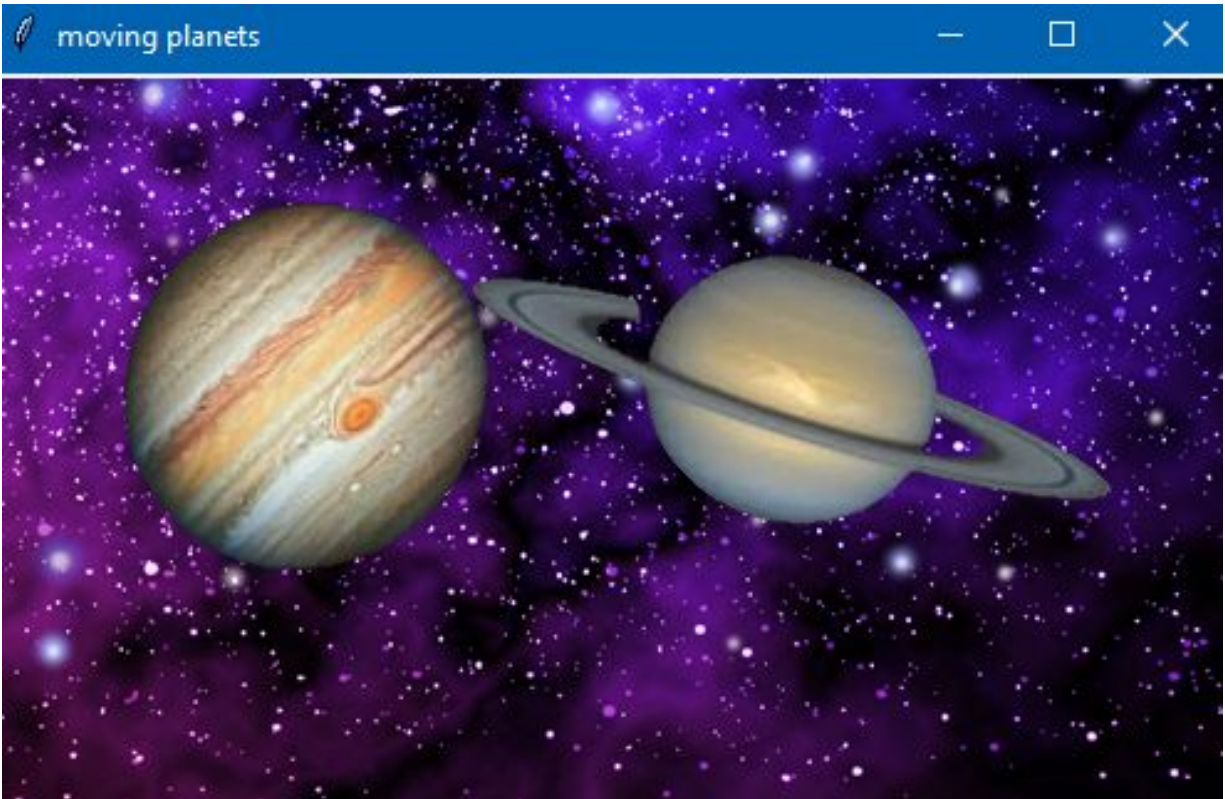
# move the planets
canvas.move(image_jup, velocity_x, velocity_y)
canvas.move(image_sat, -velocity_x, velocity_y)
win.update()
time.sleep(0.01)
t1 = time.time()
if t1-t0 > 5:
    break

win.mainloop()
```

Which gives this moving result:







This was a very short snip of an animation, but here are the general rules for all similar GUI's (like pygame):

1. The images and sounds are usually stored in a sub-folder called "assets".
2. All images are squares which have transparent backgrounds. The squares are defined by the top left corner, so to get the centre we would need to obtain the image width & height and divide by 2.
3. We use the centres and widths to determine collisions with other objects and also the screen edges (in the example code, the planets simply disappear off the screen).
4. There is always a loop, which is usually a while loop.
5. We keep redrawing the images in their new positions before updating the loop `win.update()` . And importantly, **this is the effect of motion.**

There are lots of things that we could do from here, for example we could assign masses to the two planets and invoke the force of gravity from each one upon the other.



The force of attraction on each planet (or particle) would be

$$F = - \frac{GMm}{r^2}$$

Where  $r$  is the distance between the two:

$$r^2 = x^2 + y^2$$

$M$  is the mass of the other planet (or if more than one planet, it would be the sum of the system of planets). With this method, not only could we do a simulation of planetary movements, but we could model entire galaxies or clusters of thousands of stars colliding.

The processing power of modern day computers, let alone high powered computers, are more than capable of producing amazing simulations. And what we would observe is chaotic motion but with broad observable and rather beautiful patterns.

It would be quite possible to generate and post reasonable code, but this would take about 200 lines of code.

## Maths

We have come a long way at this point and have looked at various techniques such as recursive functions, backtracking and binary trees which used classes .

We then touched upon some data science concepts, reading and writing data, visualisation of the data and finally some machine learning techniques like decision trees and  $k^{\text{th}}$  nearest neighbour.

We also looked at some fun problems which let us use a whole variety of basic techniques up to that point and implement the basic types such as strings, integers, lists, dictionaries.

The purpose of this section is to look into the mathematics of how we might consider approaching and solving some more fundamental maths problems and how this ties into the computation efficiently using python.

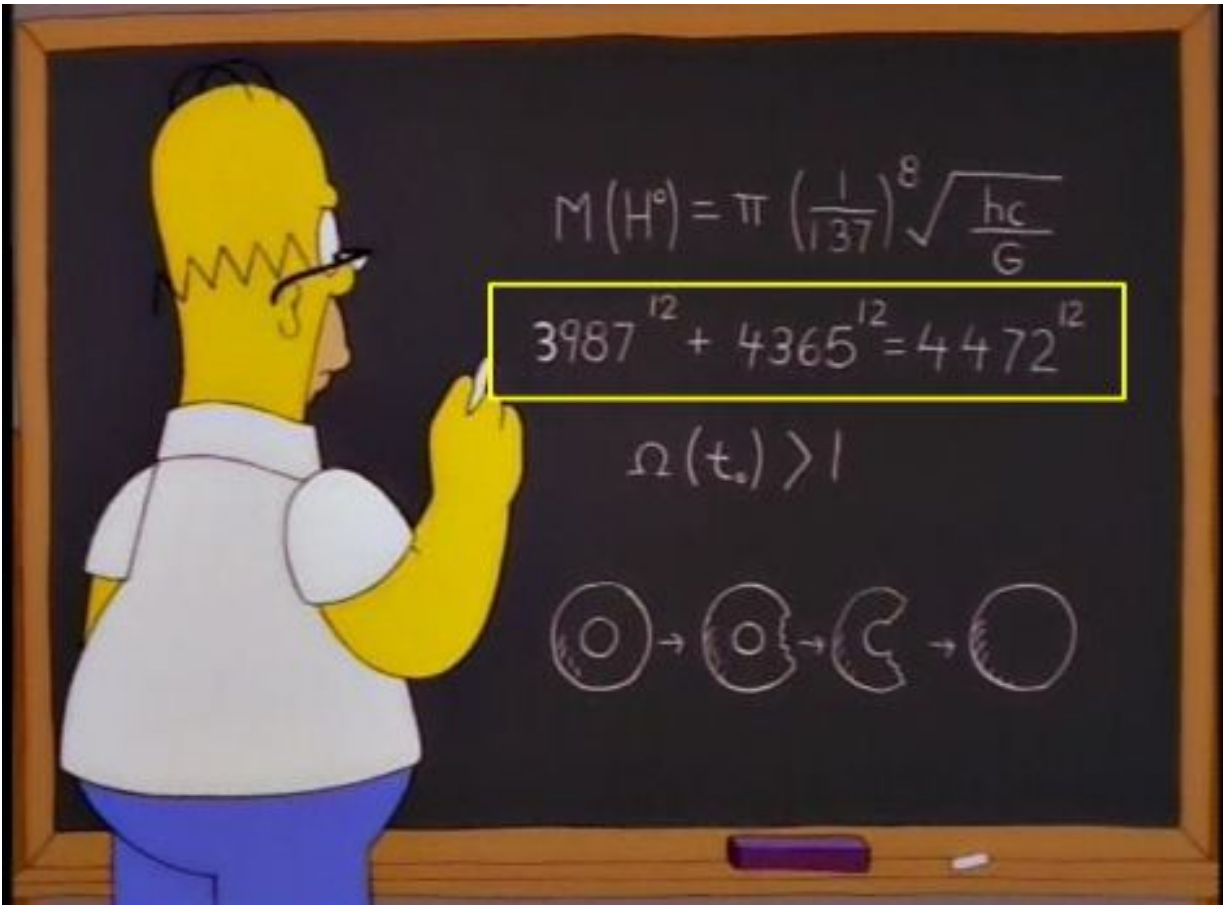
### Fermat's last theorem

In number theory, Fermat's Last Theorem states that no three positive integers  $a$ ,  $b$ , and  $c$  satisfy the equation:

$$a^n + b^n = c^n$$

for any integer value of  $n$  greater than 2.

However, in the popular cartoon series, Homer Simpson finds an exception to the case that defeats the calculators (and there are other numbers that do the same too).



If the user puts these numbers into a calculator, then they find an exception to Fermat's last theorem.

However, on inspection we find this:

```
x = pow(3987, 12)
y = pow(4365, 12)
z = pow(4472, 12)

if x + y == z: print('this works')
else: print('this fails')

print('sum of squares:', x + y)
print('the big square:', z)
```

Which gives these enormous numbers:

```
sum of squares: 2541210258614589176288669958142428526657
the big square: 2541210259314801410819278649643651567616
```

And we can actually find where the numbers differ, but adding this piece of code:

```
first_number = str(x+y)
second_number = str(z)

for i, v in enumerate(first_number):
    if second_number[i]==v: pass
    else:
        print(f'the break is at {i} which gives {second_number[i]} and {v}')
        break
```

So Fermat's last theorem was true after all. And we were able to find out where and why the old handheld calculators failed.

The rounding of large numbers in old calculators is expressed as  $6.39766563e+43$  and numbers with accuracy above the 8th decimal place will be lost.

```
the break is at index 10 which gives 8 and 9
```

Notice a couple of tricks that were done.

- We can iterate through a string as if it was a list of characters.
- Enumerate : gets the **index** and **values** of a list (analogous to the key:value pair of a dict).
- We can terminate loops early with a `break` statement.

## Complex numbers

Python naturally handles complex numbers. In fact, the `complex` type exists in the same way that `int` and `float` do.

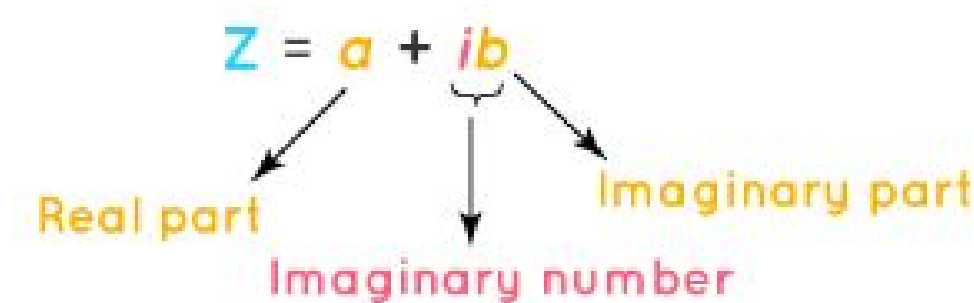
Python also provides other related functions with the `cmath` module. For a general Python object `x`, `complex(x)` delegates to `x.__complex__()`.

Complex numbers often come up in engineering, maths and physics problems and python provides useful tools to handle them.

The common form of a complex number,  $Z$ , is this:

$$z = a + i \cdot b$$

Where  $a$  is called the *real* part,  $b$  is the *imaginary* part.



The term imaginary<sup>[20]</sup> is rather confusing in the sense that it makes numbers sound quasi-magical to newcomers of the subject. The reality is that the imaginary part is an orthogonal axis to the real number line. And if we consider imaginary numbers to be **real orthogonal numbers**, then the per-se “*magic or imaginary concept*” is removed.

The numbers  $a$  and  $b$  are real numbers, or in mathematical language we say:  $a, b \in \mathbf{R}$  which means that  *$a$  and  $b$  belong to the set of real numbers*.

Numbers are an abstract concept. They are created to serve a purpose for human needs and as such have evolved over time as the human understanding of the world has evolved. Whilst this is rather philosophical for a coding book, it is useful to gain an insight into numbers and therefore part of their history and evolution to understand the concept of complex numbers and their necessity.

### Integers:

A raw need to count with fingers for example. Count apples, oranges or stones.



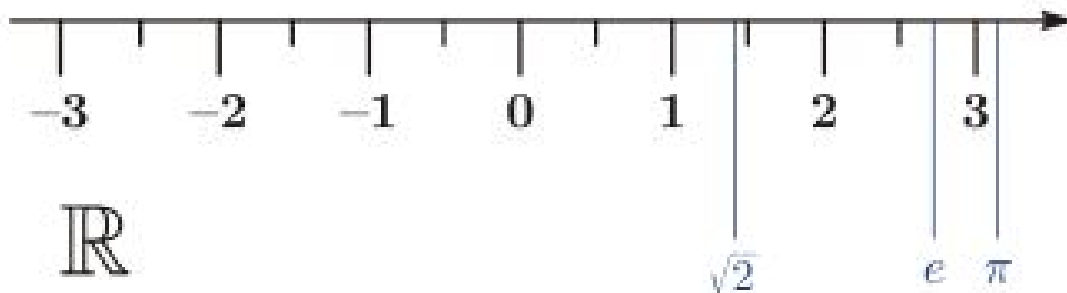
## Negative integers and Zero:

A ledger or balance and the concept of nothing (zero) requires both **negative** numbers and **zero**. In its most basic sense, negative numbers are necessary for sharing or taking.



## Real numbers:

Real numbers are all the numbers in between the integer numbers. If we break 1 into two parts then we have  $\frac{1}{2}$ . The square root of two is a number that exists between 1 and 2 and very close to 1.41. These numbers definitely exist, but are not the integers that we count on our fingers or have invented on the conventional number line.



## Finally, Complex numbers:

Complex numbers came into creation some 400 years ago. They were the roots of cubic and quadratic equations. In particular it was shown that the roots of a quadratic equation of the form  $ax^2 + bx + c = 0$  are:

$$x = -b \pm \frac{\sqrt{b^2 - 4ac}}{2a}$$

Most importantly for complex numbers is the case where:

$$b^2 < 4ac$$

It is at this point that we are taking the square root of a negative number. And without the invention of a new type of number or a change in the accepted numbering system, this simply could not be done.

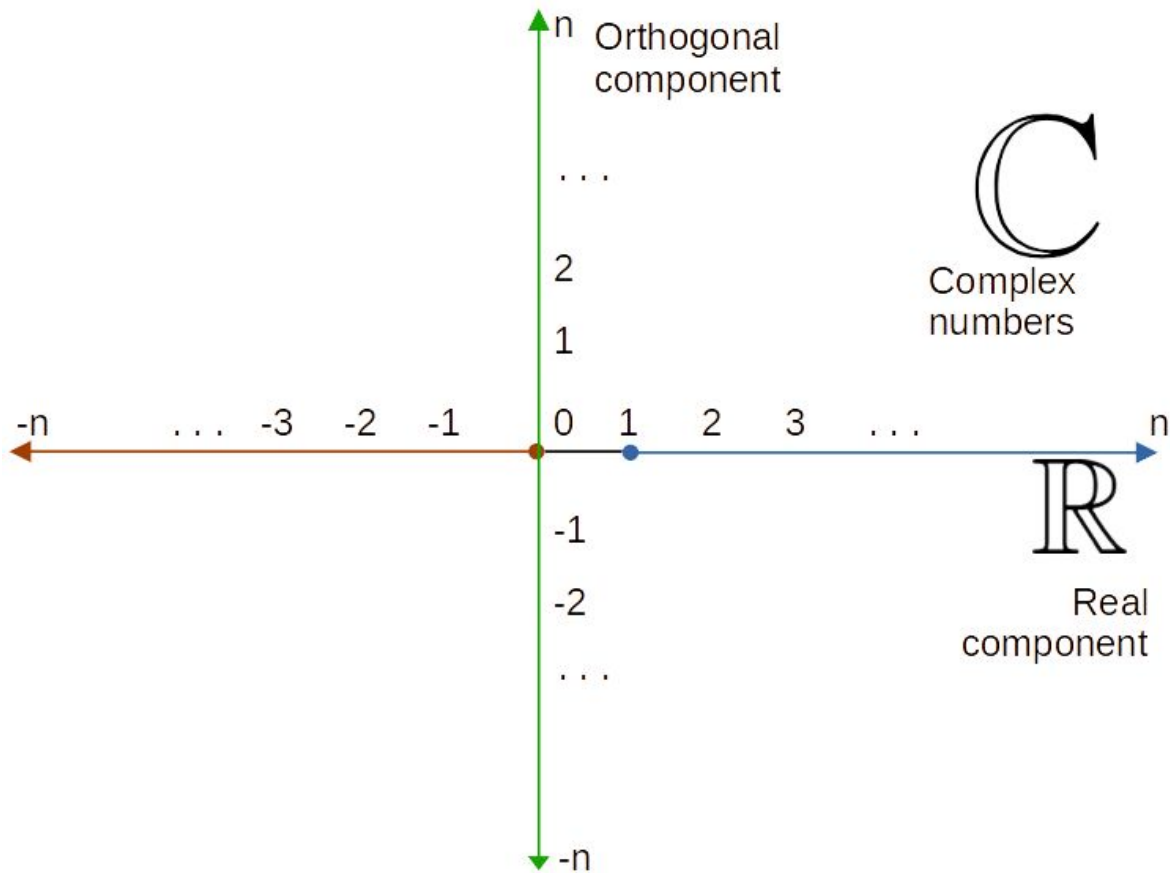
So a new number was reluctantly created and given the name imaginary,  $i$ . And the definition of  $i$  is as follows:

$$i^2 = -1$$

This means that “ $i$ ” takes two values in the same way that all square roots do as follows.

$$i = \pm \sqrt{-1}$$

And the picture of the orthogonal component to the number line, which is referred to as imaginary numbers, was born. The complex numbers, which are often denoted with a  $Z$  or a  $C$  are combinations of pure imaginary (real orthogonal) and real numbers.



At this point, we have gained sufficient understanding to proceed with code.

```
x = 5 # the real component
y = 3 # the orthogonal component

z = complex(x,y) # construct a complex number

print(z)
```

Returns this:

```
(5+3j)
```

The first thing to note is that, in computing, the letter *j* is commonly used to denote the orthogonal component (ie. the imaginary component) in place of *i*. The reason that the letter *j* is used is because in the fields of engineering and physics, the letter *i* is often used to represent current.



We can create a complex number without the need for the `complex` constructor like this:

```
z = 10 + 20j
print(type(z))
```

And this returns the type:

```
<class 'complex'>
```

So this fits naturally into the Python framework.

Complex numbers also fit into the popular `numpy` module along with many other modules like `math` and `matplotlib`. For `numpy` it looks like this:

```
import numpy as np

c = 10 + 1j

print(f'real part: {np.real(c)}, orthog part: {np.imag(c)}')
```

Which returns this:

```
real part: 10.0, orthog part: 1.0
```

We incidentally note how the components are both of type float and that a number precedes the `j` to differentiate it from a variable.

To illustrate the point, we can do this:

```
j = 10

c = 10j
d = 10*j

if c==d: print('c is the same as d')
else: print('c is different to d')

# prints: c is different to d
```

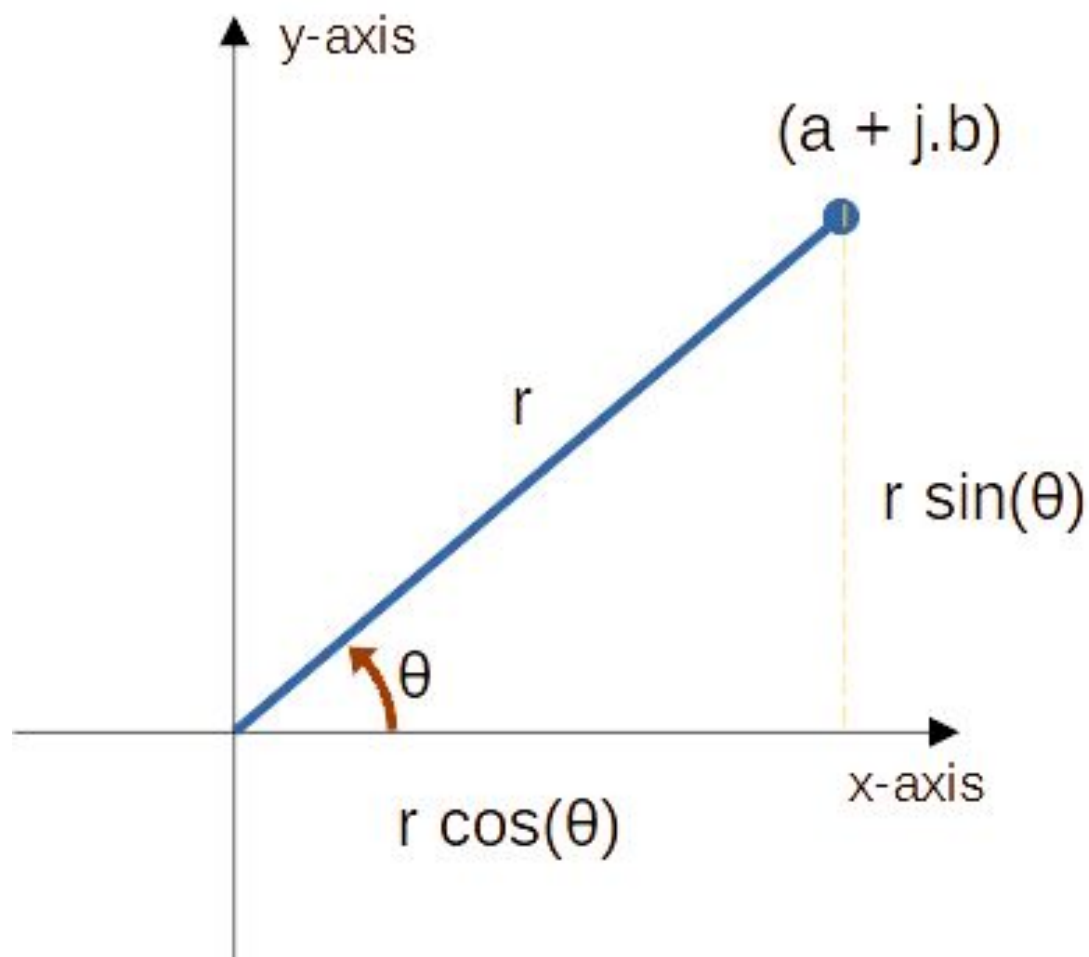
A complex number has a modulus and an argument. The modulus is defined as:

$$|z| = |a + jb| = \sqrt{a^2 + b^2}$$

Python understands this within the context of the `abs()` function, so we can do this.

```
z = 3 + 4j
print(abs(z))
# prints 5
```

The argument of a complex number is defined as the angle that it makes with the horizontal (or real) axis in the anticlockwise direction.



We can obtain the argument in two ways:

```
import cmath
import math
```

```

z = 4 + 3j

# Using cmath module
p = cmath.phase(z)
print('cmath Module:', p)

# Using math module
p = math.atan(z.imag/z.real)
print('Math Module:', p)

```

Which both return the same value:

```

cmath Module: 0.6435011087932844
Math Module: 0.6435011087932844

```

Note that these numbers are in radians so one needs to convert to degrees if this is desired using the following factor:

$$2\pi \cdot \text{radians} = 360^\circ$$

$$1 \text{ radian} = \frac{180^\circ}{\pi} \sim 57^\circ$$

Or by using a function like this:

```

import math

z = 2 + 2j
p = math.atan(z.imag/z.real)
print(math.degrees(p)) # returns 45.0

```

Finally, we show how complex vectors can be visualised in a polar coordinate plot (argand<sup>[21]</sup> diagram).

```

import numpy as np
from matplotlib import pyplot as plt

```

```

def plot_polar_vector(c, label=None, color=None, start=0, linestyle='-'):
    # plot line in polar plane
    line = plt.polar([np.angle(start), np.angle(c)], [np.abs(start), np.abs(c)],
                     label=label, color=color, linestyle=linestyle)

    # plot arrow in same color
    this_color = line[0].get_color() if color is None else color
    plt.annotate("", xytext=(np.angle(start), np.abs(start)),
                 xy=(np.angle(c), np.abs(c)),
                 arrowprops=dict(facecolor=this_color, edgecolor='none',
                                 headlength=12, headwidth=10, shrink=1, width=0))

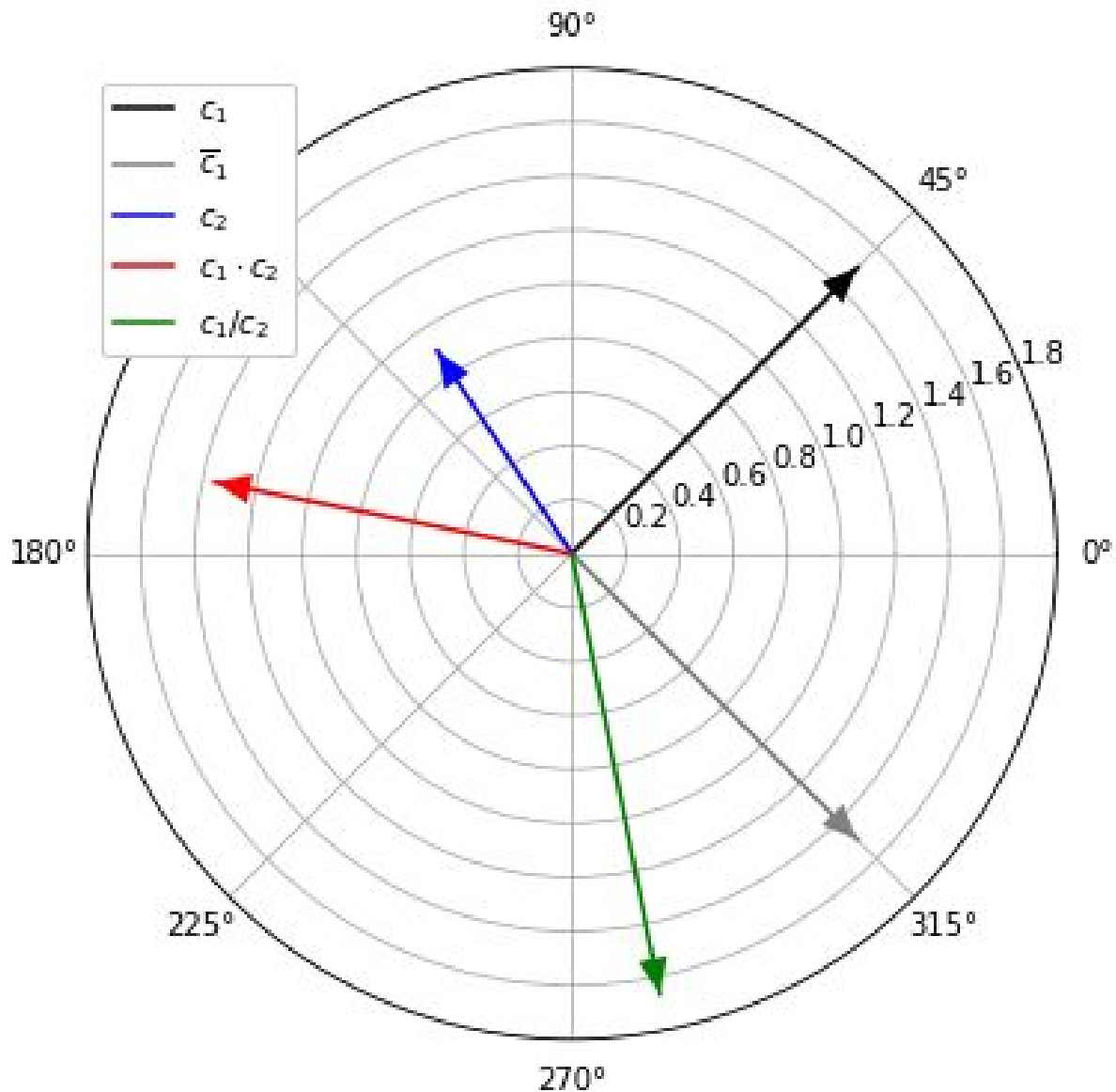
c_abs = 1.5
c_angle = 45 # in degree
c_angle_rad = np.deg2rad(c_angle)
a = c_abs * np.cos(c_angle_rad)
b = c_abs * np.sin(c_angle_rad)
c1 = a + b*1j
c2 = -0.5 + 0.75*1j

plt.figure(figsize=(6, 6))
plot_polar_vector(c1, label='$c_1$', color='k')
plot_polar_vector(np.conj(c1), label='$\overline{c}_1$', color='gray')
plot_polar_vector(c2, label='$c_2$', color='b')
plot_polar_vector(c1*c2, label='$c_1 \cdot c_2$', color='r')
plot_polar_vector(c1/c2, label='$c_1/c_2$', color='g')

plt.ylim([0, 1.8])
plt.legend(framealpha=1)

```

Which produces this:



## Parametric equations

Parametric equations are groups of equations that depend on the same parameters:  $x = f(a,b)$ ,  $y = g(a,b)$ . Quite often, in physics, the common parameter is time, so we have a position  $(x,y)$  which is a function of time:  $x = f(t)$ ,  $y = g(t)$ , but  $x$  and  $y$  are different functions of time because they have different forces acting upon them.

Another example is cartesian and polar coordinates. While the equation of a circle in Cartesian coordinates can be given by  $r^2 = x^2 + y^2$ , the parametric equations for the same circle are given by:

$$x = r \cdot \cos \Theta$$

$$y = r \cdot \sin \Theta$$

There is a single parameter,  $\Theta$ , in our example for which  $x$  and  $y$  are both functions of. This happens to be very convenient when moving between different coordinate systems and in a way we have seen this behaviour with the representation of complex numbers which exist on a surface.

We can show that both coordinate systems are effectively the same by substituting the two parametric polar equations directly back into the cartesian equivalent.

$$r^2 = x^2 + y^2$$

$$r^2 = r^2 \cdot \cos^2 \Theta + r^2 \cdot \sin^2 \Theta$$

$$r^2 = r^2 \cdot (\cos^2 \Theta + \sin^2 \Theta)$$

$$r^2 = r^2$$

So let's try out a parametric equation in python. For example an equiangular spiral. If we set the radial coordinate,  $r$  to be:  $r = ae^{bt}$  where  $a$  and  $b$  are real constants, then the two equations in their parametric forms will be.

- $x(t) = ae^{bt} \cos(t)$
- $y(t) = ae^{bt} \sin(t)$

We can plot a family of these curves in python using matplotlib.

```
import math
import matplotlib.pyplot as plt

a = 1
b1, b2, b3 = 0.1, 0.2, 0.4
times = [x/20 for x in range(0,200,1)]
```

```

x1, y1 = [], []
x2, y2 = [], []
x3, y3 = [], []

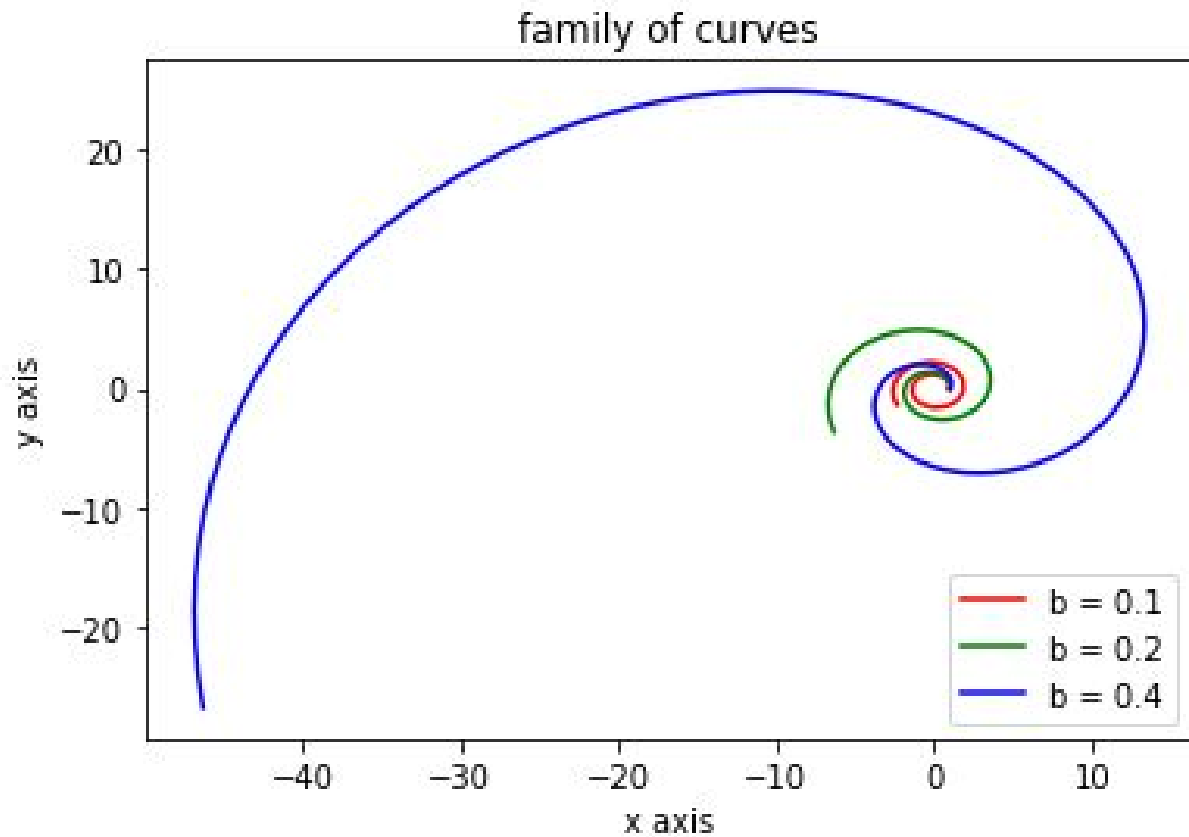
for t in times:
    x1.append(a*(math.e**(b1*t)) * math.cos(t))
    y1.append(a*(math.e**(b1*t)) * math.sin(t))
    x2.append(a*(math.e**(b2*t)) * math.cos(t))
    y2.append(a*(math.e**(b2*t)) * math.sin(t))
    x3.append(a*(math.e**(b3*t)) * math.cos(t))
    y3.append(a*(math.e**(b3*t)) * math.sin(t))

fig, ax = plt.subplots()
ax.set_title('family of curves')
ax.set_xlabel('x axis')
ax.set_ylabel('y axis')

ax.plot(x1, y1, c='red', label='b = 0.1')
ax.plot(x2, y2, c='green', label='b = 0.2')
ax.plot(x3, y3, c='blue', label='b = 0.4')
plt.legend()
plt.show()

```

Which gives this:



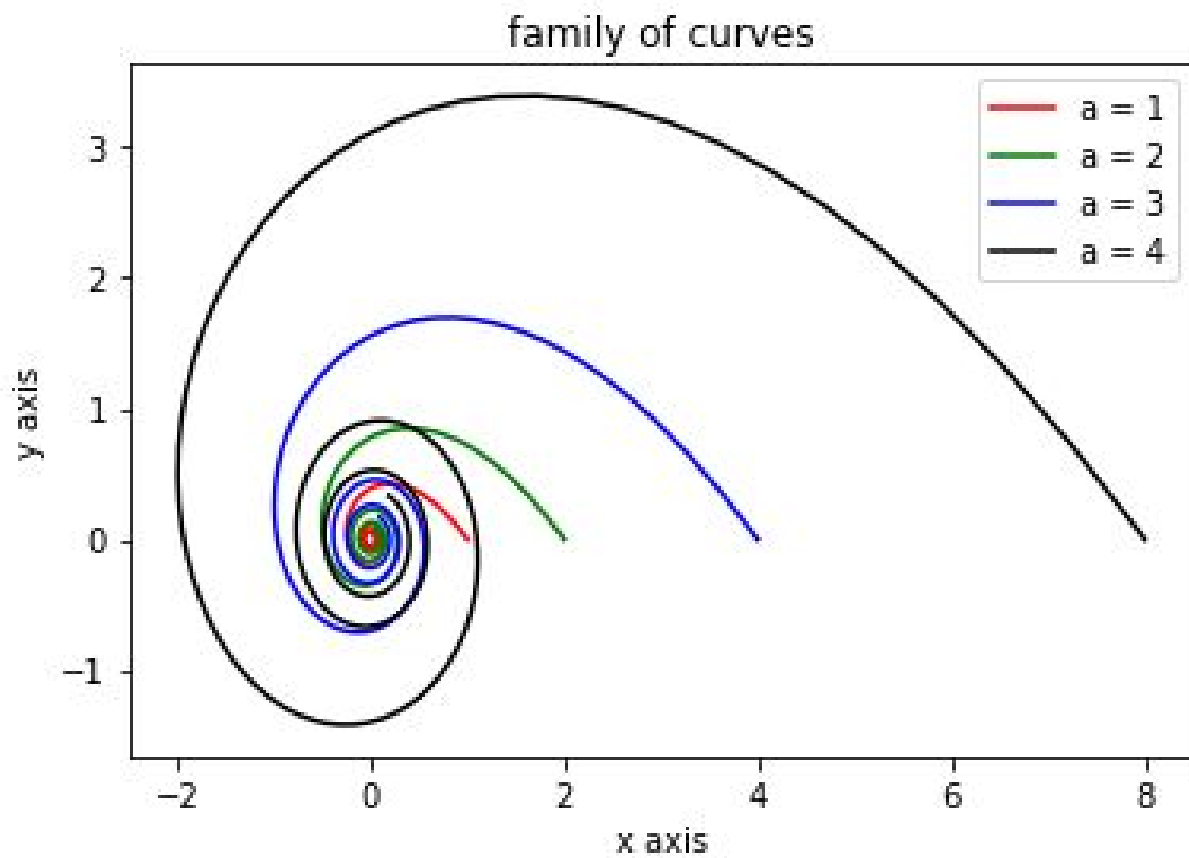
We can easily inspect other curves of the same nature. For example let's change, the radial component to be:

$r = \frac{a}{t+1}$ , where  $a$  is a constant. We adjust the code.

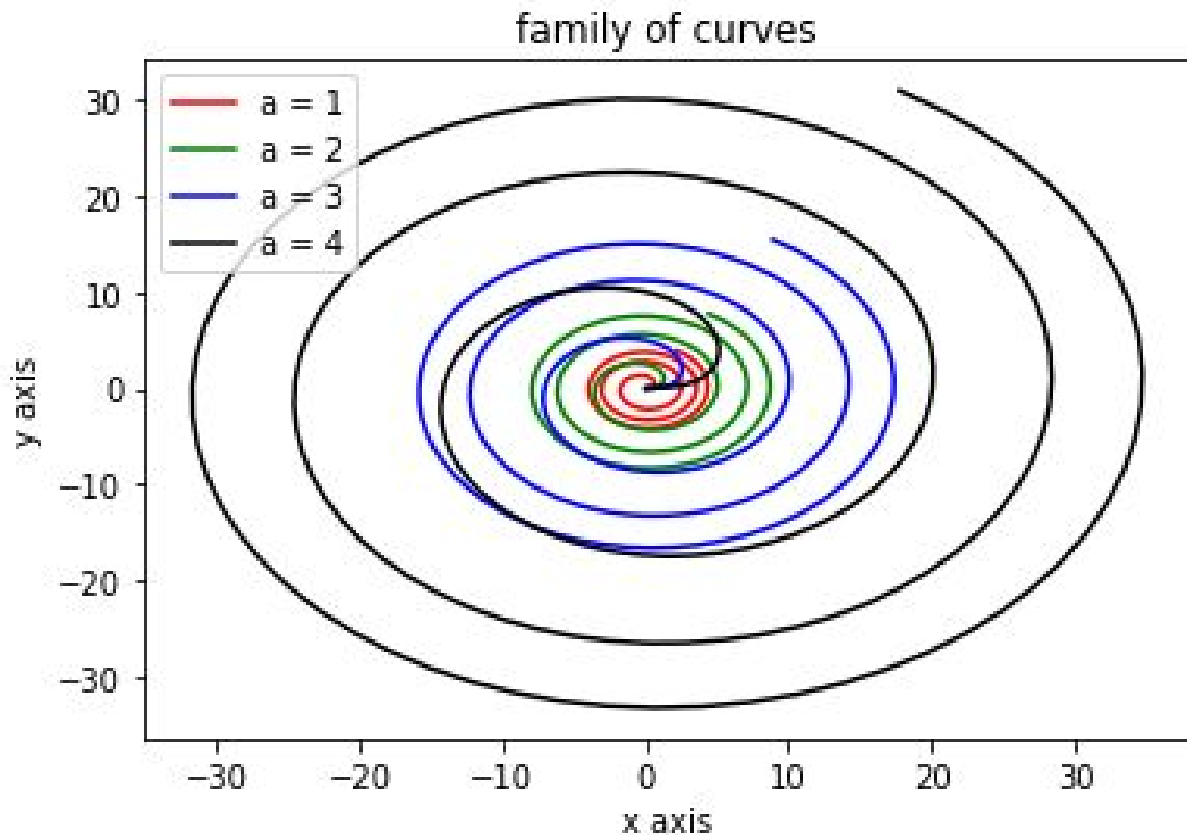
```
a1, a2, a3, a4 = 1, 2, 4, 8
```

And we get this:





Or if we set  $r = \sqrt{t}$  we get this:



The adjustment in the python code is relatively straightforward. We just had to amend the radial component of the function which occurs in the for loop to be whatever we wanted.

```
for t in times:
    r = math.sqrt(t) # radial component
    x1.append(a1*r * math.cos(t))
    y1.append(a1*r * math.sin(t))
```

Finally, by adding a third component we would create variants of the helix in a 3d plot. Adding the 3rd dimension entails just 3 steps:

1. Adding an extra list to hold the z components.
2. Populating the z components.
3. Specifying a 3d plot.

Here we add the extra list:

```
x1, y1, z1 = [], [], []
```

Here we add the z values:

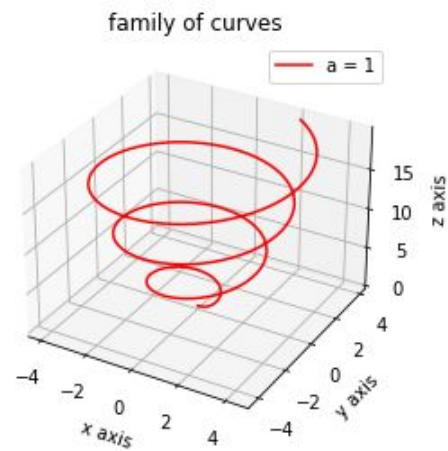
```
for t in times:
    r = math.sqrt(t) # radial component
    x1.append(a1*r * math.cos(t))
    y1.append(a1*r * math.sin(t))
    z1.append(t)
```

And here we call the new 3D plot:

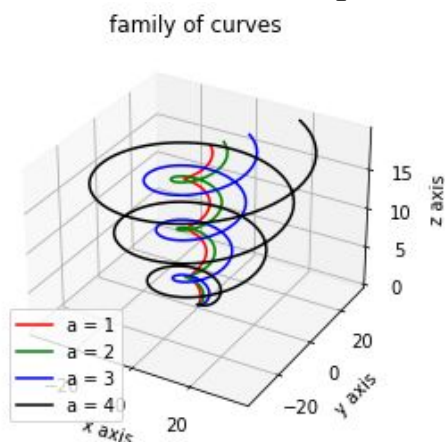
```
fig = plt.figure()
ax = plt.axes(projection='3d')

ax.set_title('family of curves')
ax.set_xlabel('x axis')
ax.set_ylabel('y axis')
ax.set_zlabel('z axis')

ax.plot3D(x1, y1, z1, c='red', label='a = 1')
```



And with this we can produce the helix family



## Derivatives

Derivatives are hugely important right across the spectrum of subjects. A derivative is essentially the rate of change of one variable with respect to another variable.

The general concept is that for any given function  $y = f(x)$ , we seek the rate of change of  $y$  with respect to  $x$  where:

$$y = f(x)$$

$$\frac{dy}{dx} = \frac{d}{dx}f(x)$$

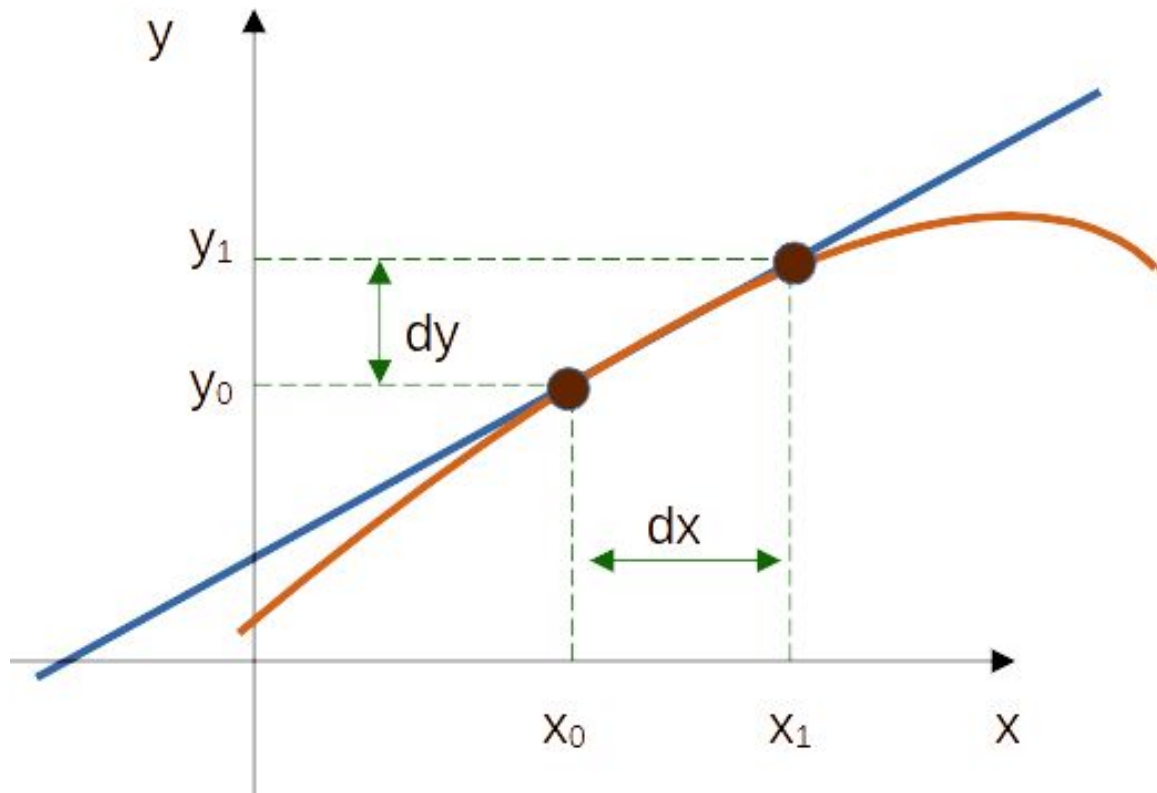
The same concept could be extended to higher dimensions, where:  $a = f(x, y, z, t)$  for example. However, the principals are still the same and in this case one might differentiate with respect to every independent variable in the function.

$$\frac{d}{dx}f(x, y, z, t) + \frac{d}{dy}f(x, y, z, t) + \frac{d}{dz}f(x, y, z, t) \dots$$

Ultimately, however, this process is done, what we are seeking is a slope between two points.

The computer version of  $\frac{dy}{dx}$  is  $\frac{\Delta y}{\Delta x}$  where:

$\Delta y = y_1 - y_0$  and  $\Delta x = x_1 - x_0$  and these are the discrete calculations that are done.



Regardless of the shape of the curve, finding the slope is a process of taking a section of the curve and computing the differences.

A curve in python might be made up of two lists. There are other ways of creating curves (for example a single list of tuples), but this is convenient.

```
x = [x for x in range(10)] # [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
y = [x**2 for x in x] # [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

We could then calculate the slopes.

```
slopes = []
for i in range(1, len(x)):
    dy = y[i] - y[i-1]
    dx = x[i] - x[i-1]
    slopes.append(dy/dx)
print(slopes)
# [1.0, 3.0, 5.0, 7.0, 9.0, 11.0, 13.0, 15.0, 17.0]
```

The first thing to note is that the length of the slopes list is 1 less than the original lists. This is because the slopes are the result of the calculation *between* any two points.

In the case of the example, we now have a descriptive calculation for the way that  $y$  varies with  $x$ . And we could interpolate values between the points.

There are a whole number of methods for interpolation, such as straight line, cubic spline or a polynomial fit each own with its merits and errors.

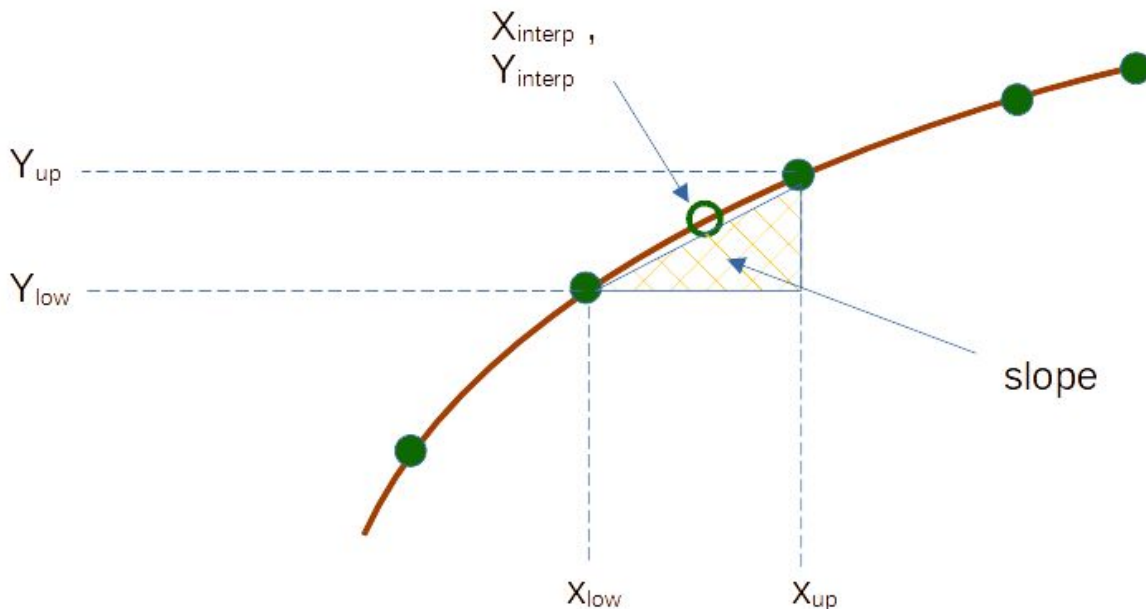
A straight line interpolation is the easiest method. It works well when the curvature of the curve that we are trying to interpolate is low or when the number of datapoints is high.

The calculation for the interpolated point is:

$$y_{\text{interp}} = x_{\text{low}} + (x_{\text{interp}} - x_{\text{low}}) \cdot \frac{y_{\text{up}} - y_{\text{low}}}{x_{\text{up}} - x_{\text{low}}}$$

$$y_{\text{interp}} = x_{\text{low}} + (x_{\text{interp}} - x_{\text{low}}) \cdot \text{slope}$$

Where  $x_{\text{low}}$  is the lower bound of the range and  $x_{\text{up}}$  is the upper bound of the range.



Following the recipe in the above picture and given that we have already computed the slopes, we are able to complete the interpolation process in

Python.

```
x_list = [x for x in range(10)]
y_list = [x**2 for x in x_list]

def interp(x_interp: float, known_xs: list, known_ys: list) -> float:
    """ give point x, return interpolated y """

    slopes = []
    for i in range(1, len(known_xs)):
        dy = known_ys[i] - known_ys[i-1]
        dx = known_xs[i] - known_xs[i-1]
        slopes.append(dy/dx)

    # find x low
    for i, v in enumerate(known_xs):
        if v < x_interp: lower_idx = i
        else: break

    y_interp = known_ys[lower_idx] + \
        (x_interp - known_xs[lower_idx])* slopes[lower_idx]

    return y_interp

y = interp(2.5, x_list, y_list)
print(y)
```

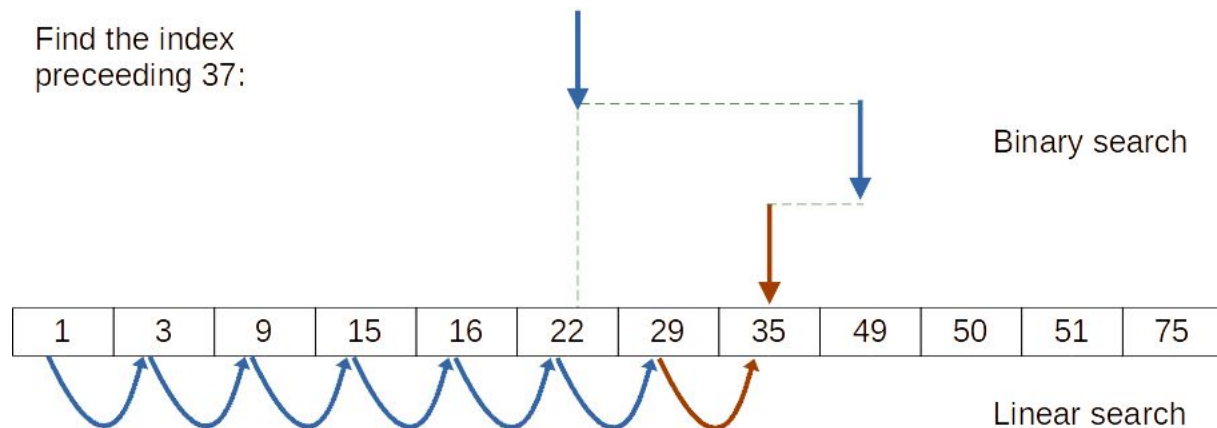
This finds the answer quickly, but there are three points to note:

- Not the fastest.
- Convexity error.
- Out of bounds: extrapolation.

**Not the fastest:** The method considered above takes a *linear* approach to locating the index of x. It simply iterates through the *sorted* list until it finds the index whose value is greater than x and then stops. The time complexity

of this is  **$O(n)$**  in computing language, meaning that the longer the list the longer the finding process takes.

A binary search (also known as half interval search or logarithmic search) using the `bisect` module in python is more optimal. The following diagram may intuitively explain why this is the case.



In particular, we see that the binary search takes fewer iterations to achieve the result and intuitively this would most certainly be the case, especially where the list was large. The time complexity for the binary search is  **$O(\log n)$**  compared to  **$O(n)$**  for the linear search.

Here are the two methods:

```
x_list = [1, 3.5, 5, 9.2, 20, 50.75]
n = 7.5

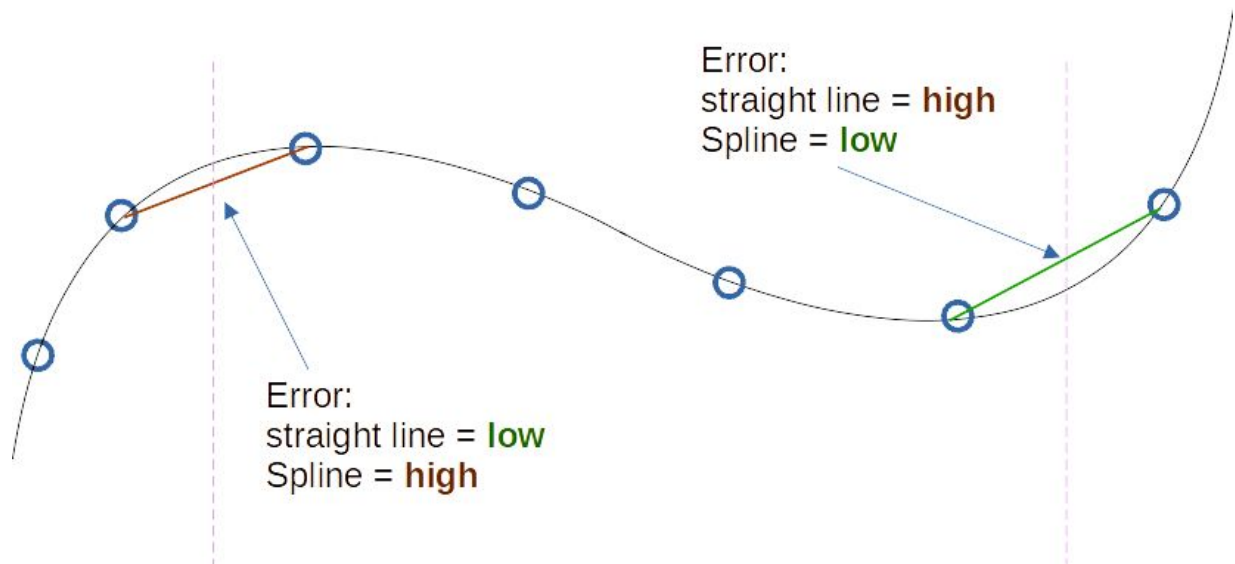
# linear search method
for i, v in enumerate(x_list):
    if v < n: r = i
    else: break
print(r)

#binary search method
import bisect
r = bisect.bisect_left(x_list, n)-1
print(r)
```



**Convexity error:** In the case of the straight line interpolation this is easier to explain, but it will also be true for other interpolation methods.

In general, whatever method is deployed, there will be an error to the true value. Some methods are better than others in certain situations, for example do we select a cubic spline or straight line ? have we overfitted the data with a cubic spline or under fitted the data with a straight line ?



Given that we know the errors exist we can change methods or make corrections using pythagoras for example. A change of methods might be by increasing the order of the polynomial used to fit the curve.

Whilst we have not talked about the mathematics of the quadratic or cubic splines, the secret to the computation is to ensure that the derivatives (slopes) of the curve at the points are equal to each other - this ensures a smoothly fitted curve.

### **Extrapolation:**

So we now have a number of options for fitting data. The two most prominent methods are straight line (simple and intuitive) and cubic spline (fits curves quite well).

But what approach do we take when the required interpolation point exists outside of the sample range.

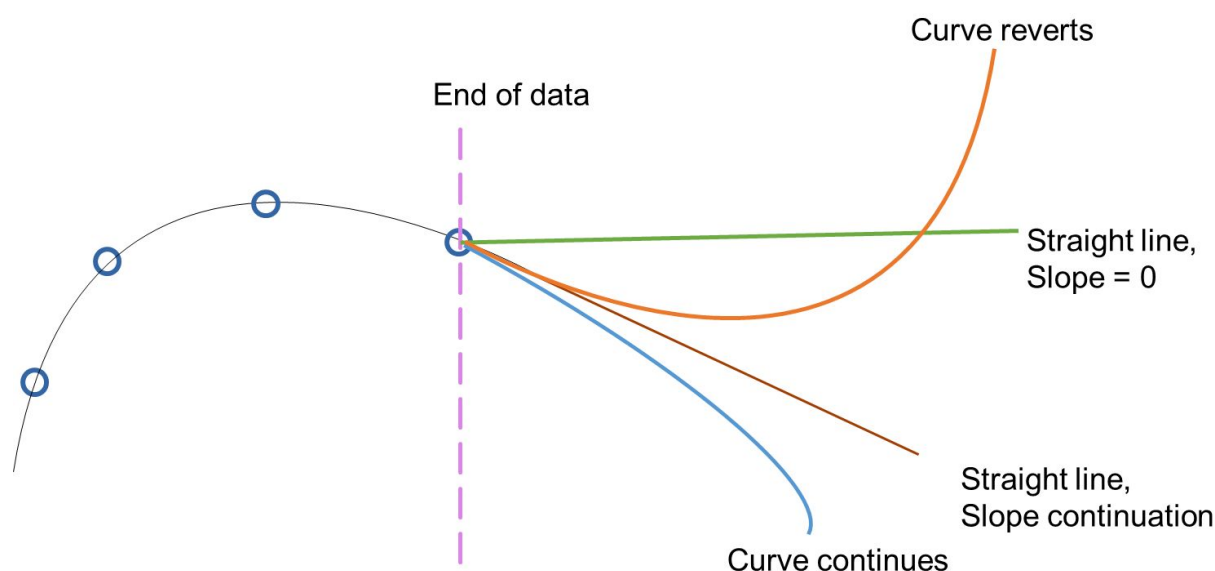
interpolation:  $x_{\min} \leq x_{\text{interp}} \leq x_{\max}$

extrapolation:  $x_{\min} > x_{\text{interp}} > x_{\max}$

Here are a few immediate options:

1. Give up: Error message that data is out of bounds.
2. All points equal to the last point.
3. Slope continuation.
4. Slope and second derivative continuation.
5. Some other cases to infinity.

We have to know how to treat the data when out of bounds in the sense of what to expect. For example, is a straight line continuous slope or another model appropriate? the extrapolation may only be suitable for a short period.



Whichever method we use, for the purpose of the code, we would only have  $x_{\text{low}}$  and  $dx$ . The final result will depend on the assumed function in the extrapolated zone.

### Maxima, minima & points of inflexion

In standard calculus we have definitions of maximums, minimums and inflexion points. For example, if we let  $f$  be a function defined on an

interval  $[a,b]$  or  $(a,b)$ , and let  $p$  be a point in  $(a,b)$ , i.e., not an endpoint, if the interval is closed.

- $f$  has a local minimum at  $p$  if  $f(p) \leq f(x)$  for all  $x$  in a small interval around  $p$ .
- $f$  has a local maximum at  $p$  if  $f(p) \geq f(x)$  for all  $x$  in a small interval around  $p$ .
- $f$  has an inflection point at  $p$  if the concavity of  $f$  changes at  $p$ , i.e. if  $f$  is concave down on one side of  $p$  and concave up on another.

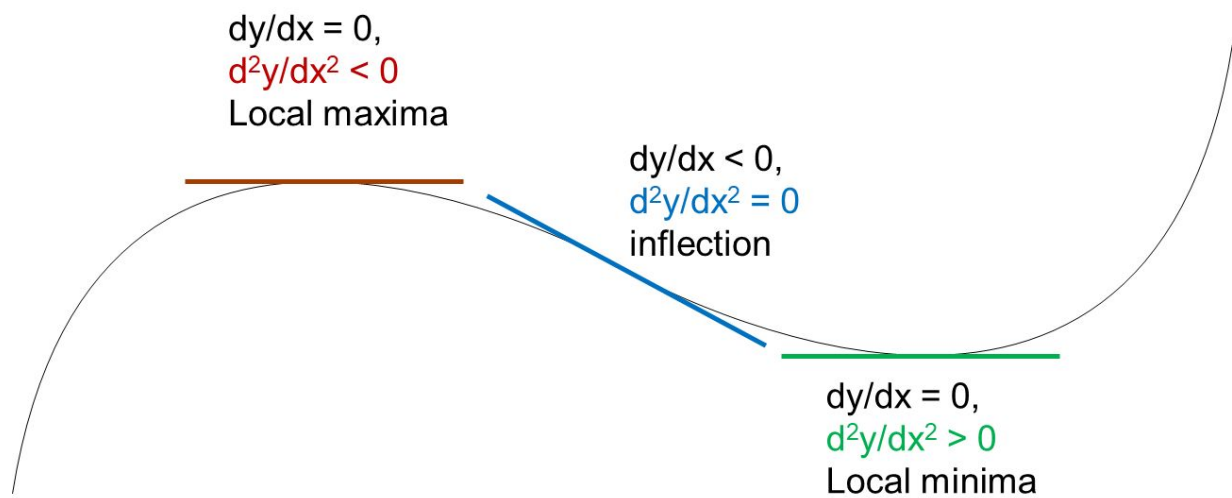
In addition to the above, we can use the first and second derivatives to understand the shape of the function and therefore determine maxima, minima and inflection points from this.

Given the function  $y = f(x)$ , we can find all points where the slope is flat (ie. zero) using the first derivative  $dy/dx = 0$ .

We can then use the second derivative to inspect how the first derivative

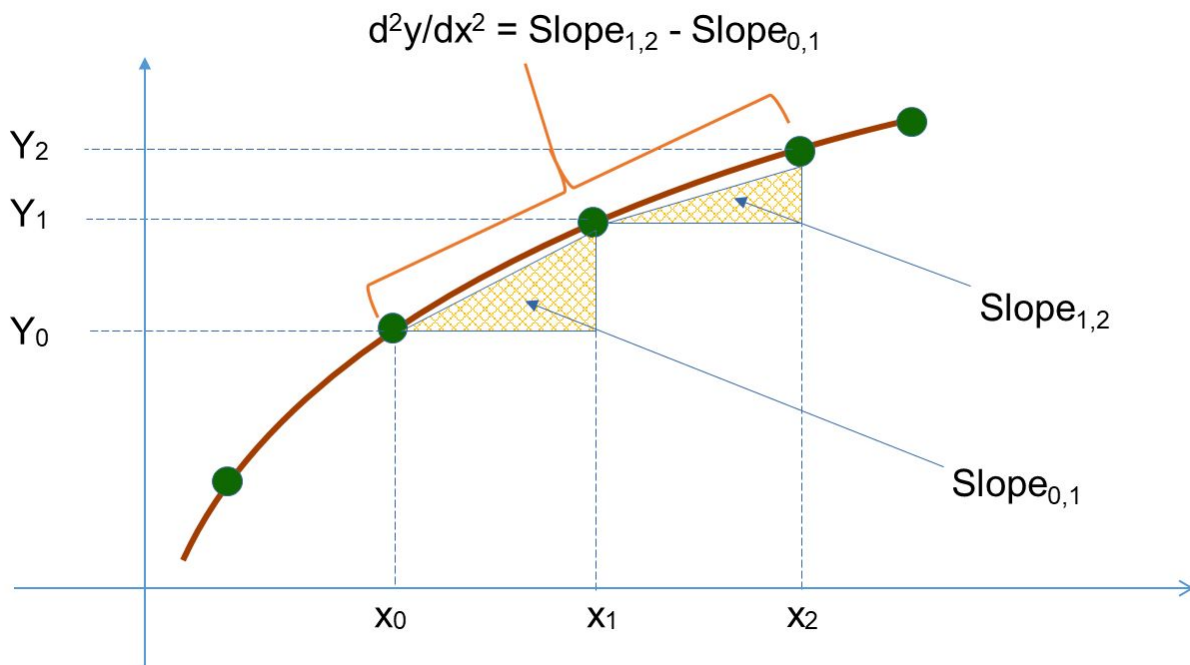
changes. So  $\frac{d^2y}{dx^2} = \frac{d}{dx}\left(\frac{dy}{dx}\right)$  tells us how the slope is changing. There are three cases:

- If  $\frac{d^2y}{dx^2} < 0$ , then the slope is continually decreasing and therefore we have a **maximum** point.
- If  $\frac{d^2y}{dx^2} > 0$ , then the slope is continually increasing and therefore we have a **minimum** point.
- And if  $\frac{d^2y}{dx^2} = 0$ , then the slope is not changing and therefore we have an **inflection** point.



We can use this type of construct effectively in Python code to determine such points, although the determination will depend on the type of data and how it is sampled<sup>[22]</sup>.

To find the slope we needed at least 2 points. And to find the change in the slope (*the second derivative*) we need to have two slopes, so therefore need at least 3 points of data.

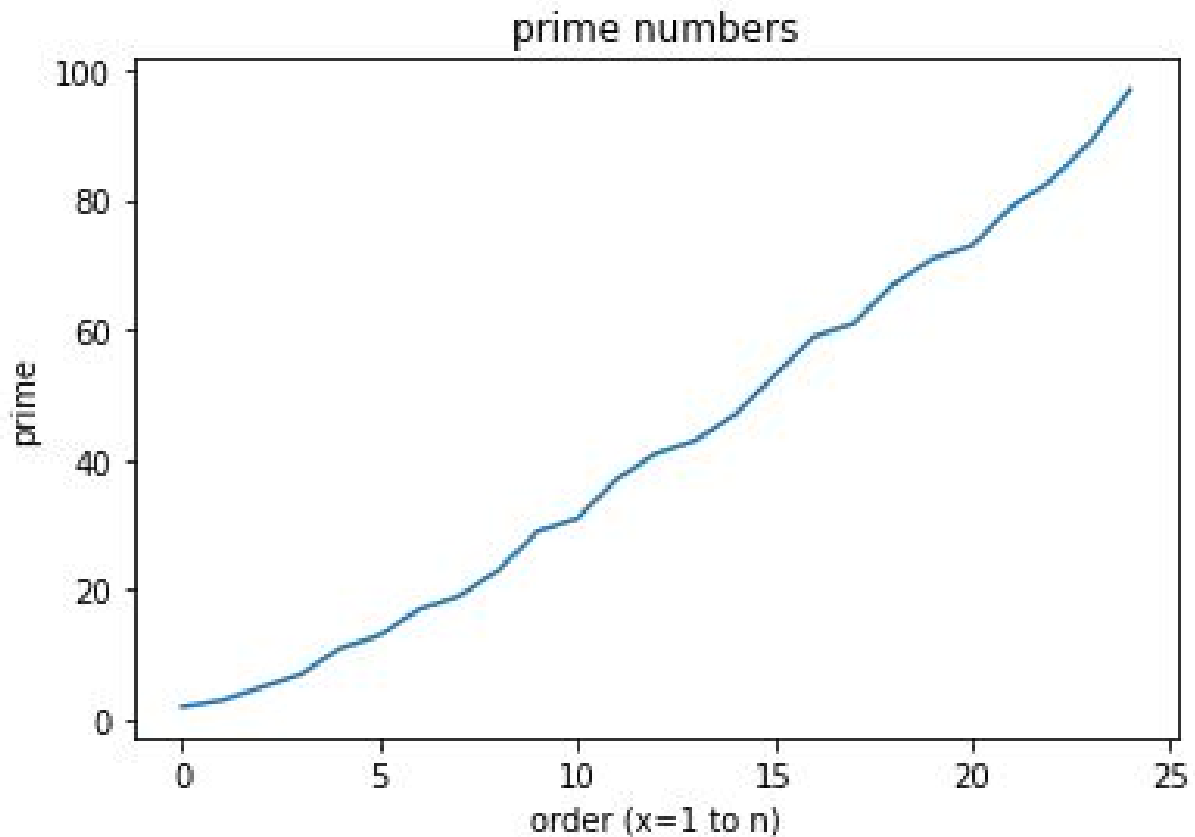


The image above simplifies what we seek to complete in order to perform the analysis.

So now let's do this in code by setting up a list of prime numbers and computing the slopes between them. This function generates primes efficiently.

```
def gen_primes(n):  
    """ Returns a list of primes < n """  
    sieve = [True] * n  
    for i in range(3, int(n**0.5)+1, 2):  
        if sieve[i]:  
            sieve[i*2::2*i] = [False] * ((n-i*i-1)//(2*i)+1)  
    return [2] + [i for i in range(3, n, 2) if sieve[i]]  
  
n = 1_000_000  
prm_list = gen_primes(n)
```

The list of primes is rather linear, with few bumps. So it will be harder detecting the slopes.



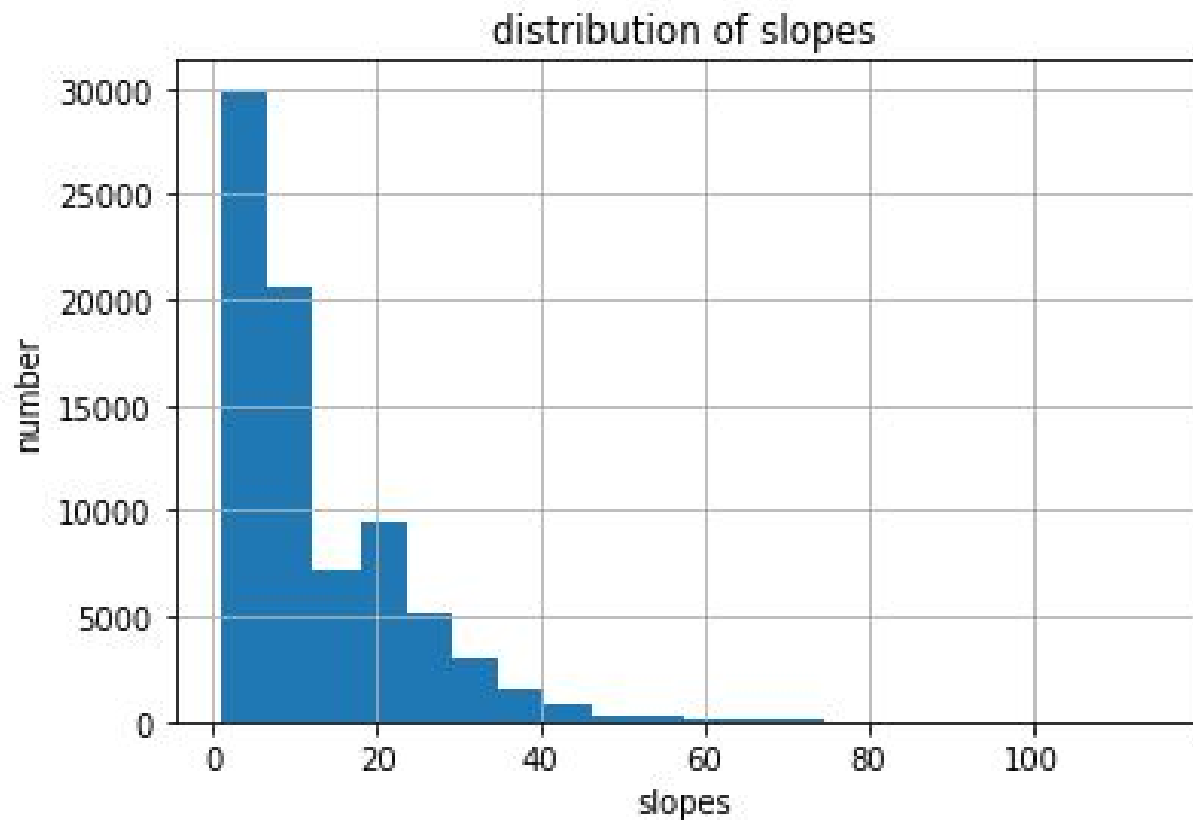
And in fact if we plotted the bigger picture, the result looks like a straight line.

We then need to compute the slopes between the primes.

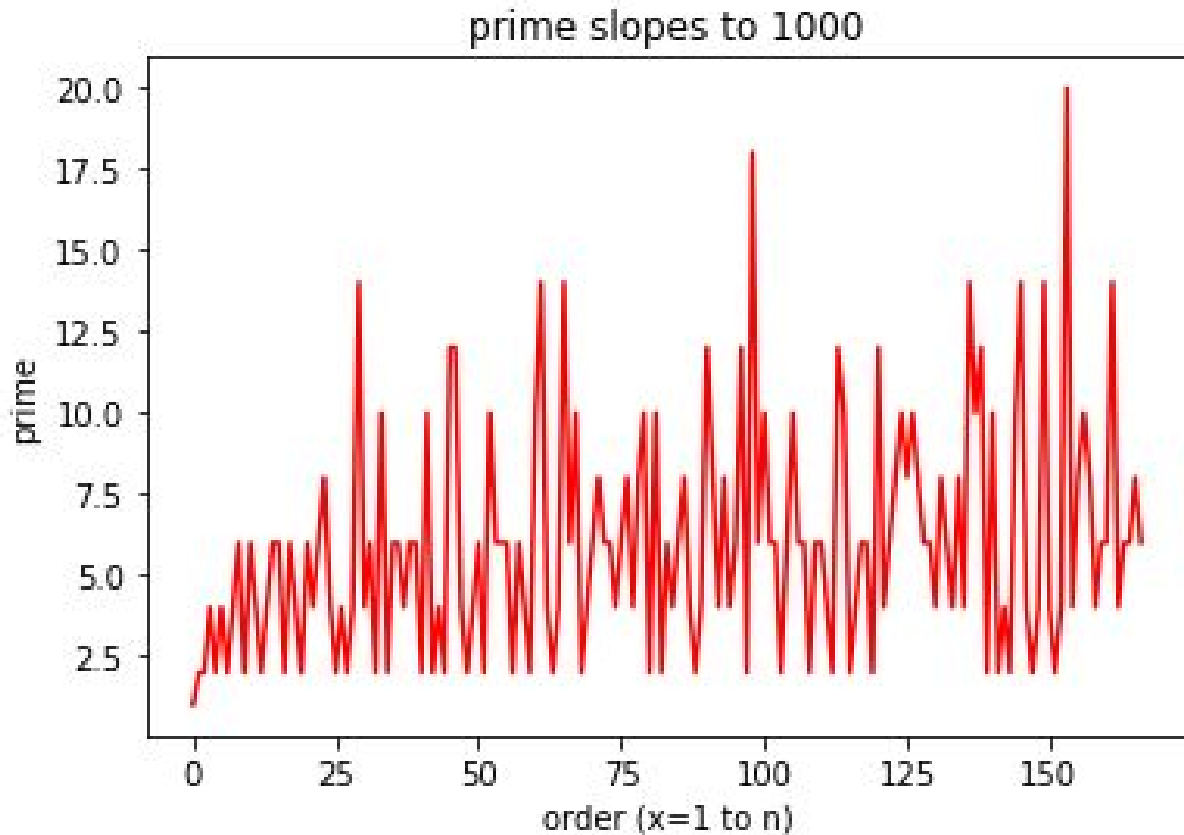
```
x_values = []
prm_slopes = [] # generate prime slopes
for i, v in enumerate(prm_list[1:]):
    prm_slopes.append(v-prm_list[i])
    x_values.append(i)
```

Notice that we were easily able to generate all primes up to 1 million in less than a second. The algorithm found 78,498 such primes, which is significantly more than is needed for the exercise.

The distribution of the slopes looks like this:

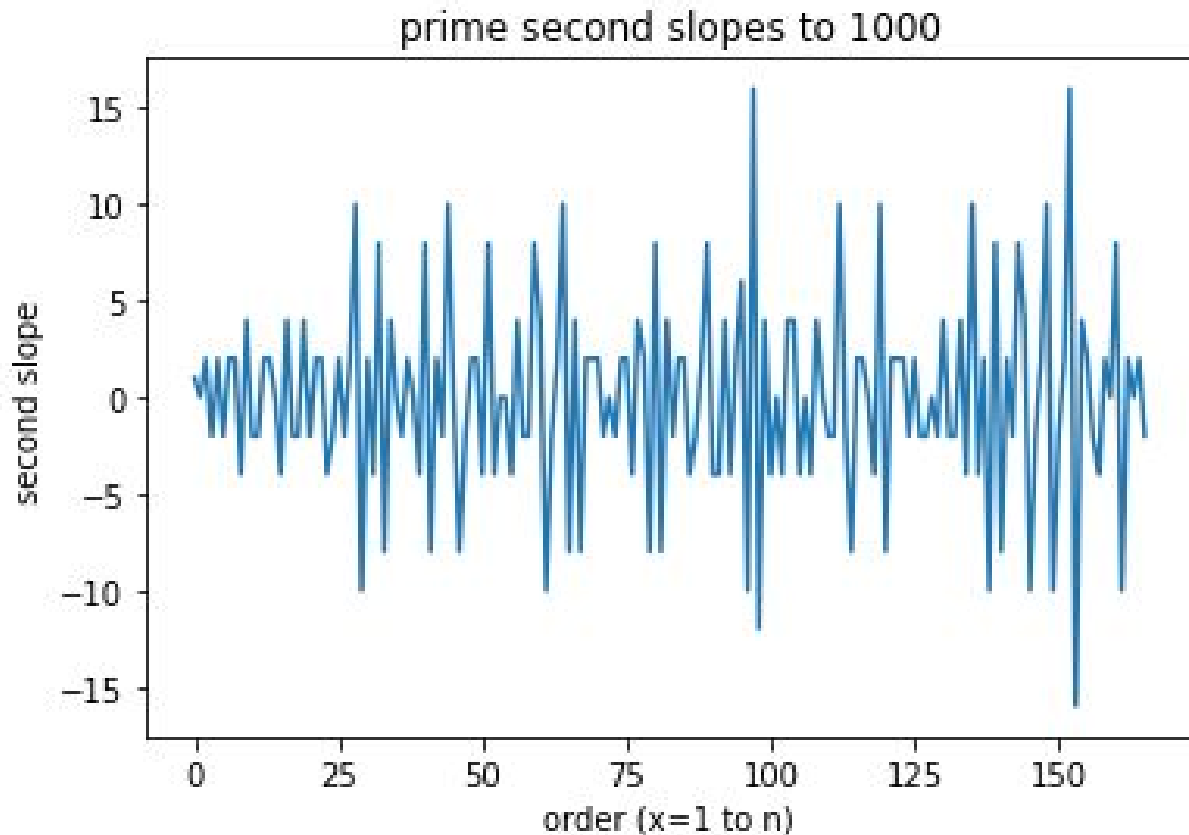


With the biggest slope between any two primes of up to 1 million being 114. The slopes are always positive because prime numbers increase in value.



We can find the second derivative in the same way that we found the first derivative. And we find this more interesting as the second derivative includes rates of change of the slopes which is the convexity at any point on the curve.

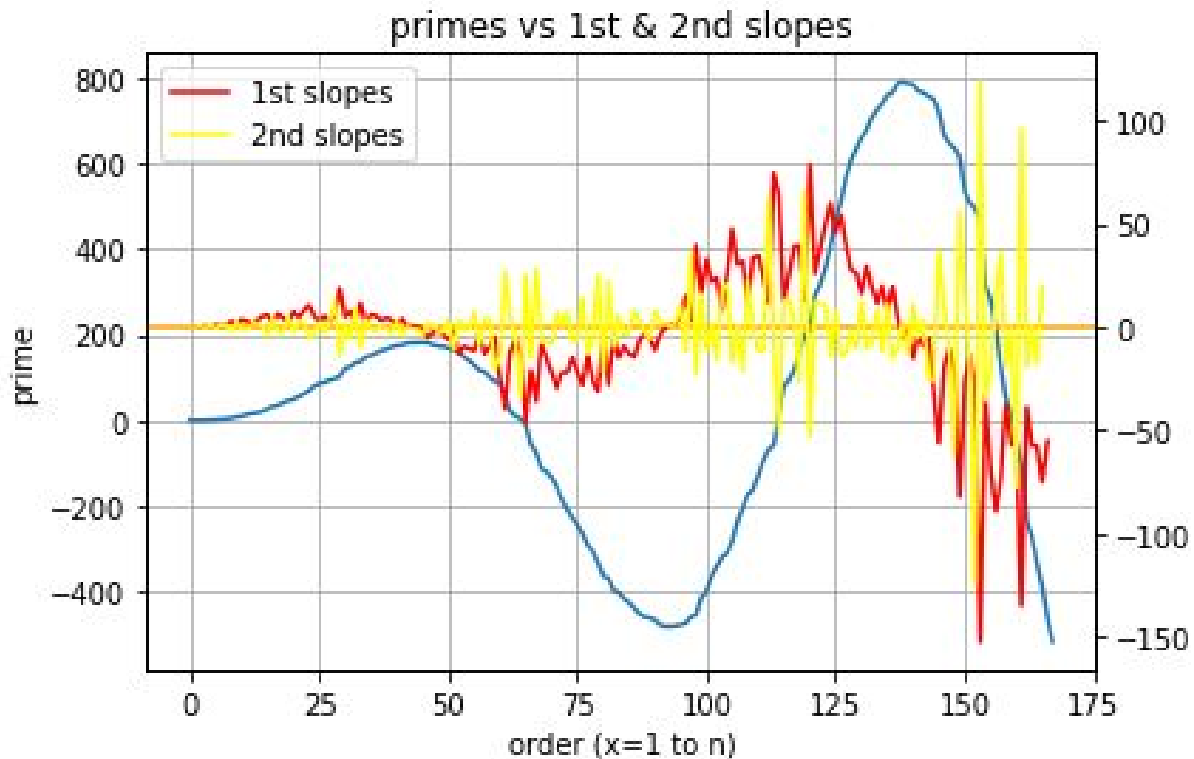




We can see that the first and second derivatives describe the curve. In this case, reporting a benign story. So let's oscillate this prime curve with a sine wave. We can do this with a single line of code:

```
prm_list = [x*math.sin(x/100) for x in prm_list]
```

Next we plot both the sin adjusted prime curve and both its 1st and 2nd slopes on a single chart.



And finally we can identify the maxima and minima (1st slope = 0) and also the points of inflection (2nd slope = 0).

In python, we are able to do this in a powerful single line of code for each list like this:

```

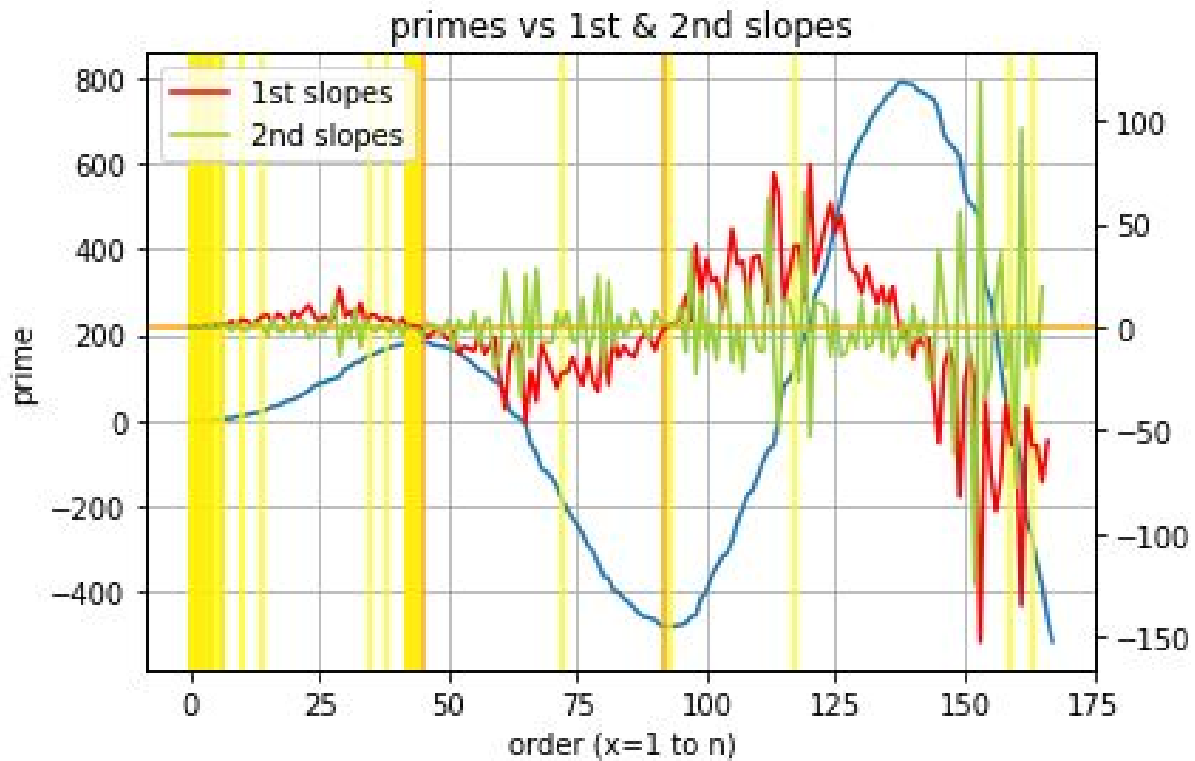
null_slopes = [i for i,v in enumerate(prm_slopes) if -1<v<1 ]

```

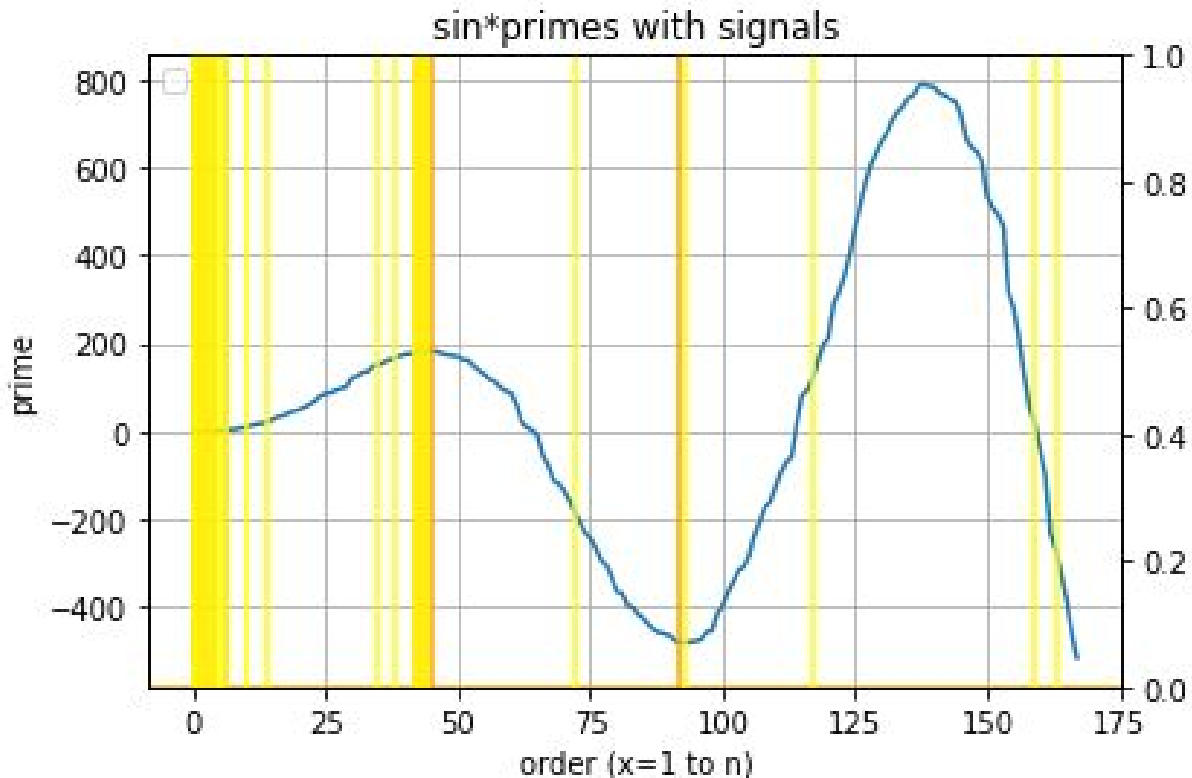
We were able to do this because of these features:

- List comprehension: `[x for x in list]`
- Enumeration unpacking: `i,v in item`
- Chained conditions: `-0.5 < v < 0.5`

And plot everything onto one chart.



In the above, we have successfully identified all the maxima and minima (with orange vertical lines) and also the inflection points (yellow lines). And we note that if we were to now remove the 1st and second slopes, we would be left with the points of interest.



Interestingly, one signal is missed at the peak of  $x = 140$  and this is because the derivative moved too fast (from +5 to -5) missing the tighter criteria of  $-1 < v < 1$ . So a good algo needs to be tweaked to catch all points of interest (in this case, we could have loosened the conditions).

In general, where the data is discontinuous in nature, such as stock prices for example, it might be beneficial to smooth the curve by generating a moving average. On the one hand this has benefits of a now smoothed set of data, but on the other hand we have lost some accuracy in the smoothing process.

This type tradeoff (a gain in  $x$  vs a detriment in  $y$ ) exists in nearly every physical system and it is for the user to find the optimal solution for any given criteria.

## Newton raphson

We looked at methods of finding solutions in a manner that is quicker than an iterative solution. For example, we were able to compare a binary search

with a linear search and show that we could reduce time complexity from  $O(n)$  to  $O(\log n)$ .

We have also looked at functions and derivatives and can numerically compute the derivative of a function at a given point by using two local points to work out the slope.

For example, given the function  $y = f(x)$ , we were able to computationally achieve  $dy/dx$  at the point  $x=n$  by taking a neighbour point  $(n+1)$  and working out the slope.

We refer to a function as  $f(x)$  and the derivative of the same function as  $f'(x)$ . We do this because the notation is cleaner (but still means the same thing  $f'(x) = df/dx$ ).

We say:  $f'(x) = \frac{d}{dx}f(x) = \frac{df}{dx}$

And:  $f'(x_n) = \left[ \frac{df}{dx} \right]_{x=n}$

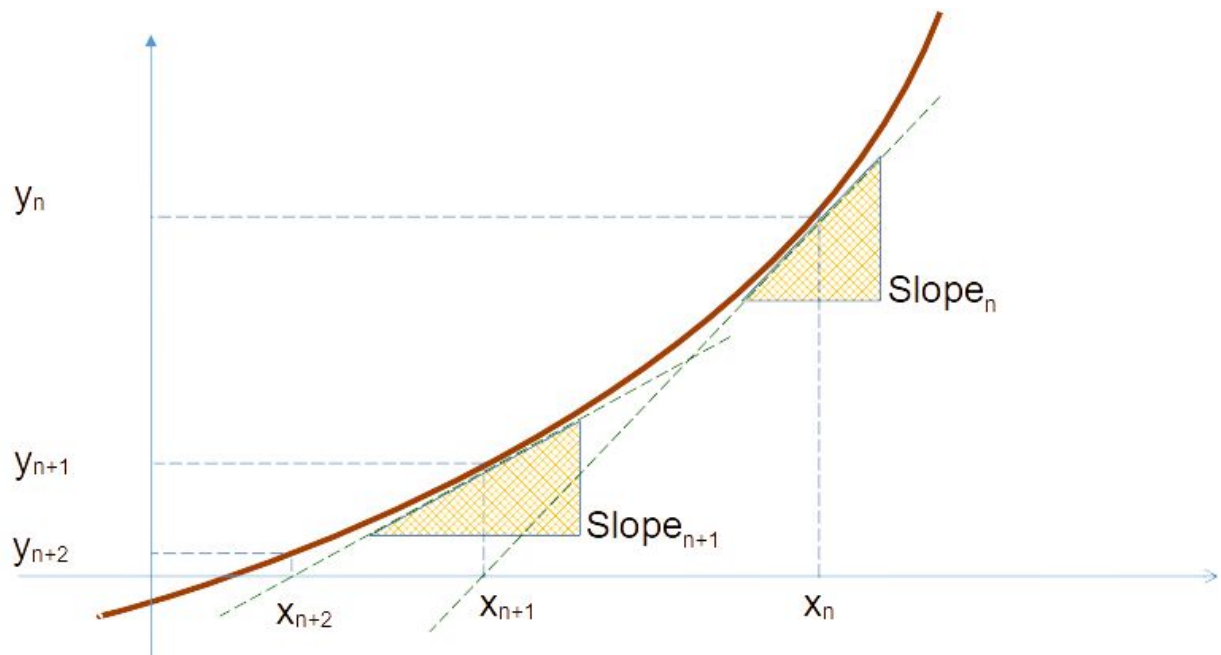
Meaning *the derivative at the point  $x=n$ .*

The newton-raphson formula is iterative in the same sense that the binary search was. Basically, this means that given an  $x_n$  we can find the next point  $x_{n+1}$  and keep on repeating this process until we have iterated a way towards the final result.

We start with the famous formula:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

The geometrical construction of the above formula looks like this:



We see that with this method, the convergence to the solution is fast and if we know the derivative of the function, that we can achieve the result quicker than both the binary search and certainly a linear iteration.

If we try this on a well known number like  $\sqrt{2}$  or any non-square integer or number, then we can observe the number of iterations that obtains a satisfactory accuracy.

Let  $f(x) = x^2 - 2$

Then  $f'(x) = 2x$

Our iterative formula becomes:

$$x_{n+1} = x_n - \frac{x_n^2 - 2}{2x_n}$$

We can implement this directly into python.

```
def function(x):
    """ function of x """
    return x**2 - 2

def derivative(x):
```

```

    """ derivative of f(x) """
    return 2*x

def newton_raphson(xn, f, df):
    """ take xn, a function and its derivative
    return x(n+1)
    """
    xn1 = xn - f(xn)/df(xn)
    return xn1

```

And we can try to implement our guesses to an accuracy of 1 part in 1 million like this:

```

# initial guess
initial_guess = 1.5

for i in range(10):
    next_guess = newton_raphson(initial_guess, function, derivative)
    if initial_guess - next_guess < 0.000_001:
        print(f'exit at loop {i} with result {next_guess}')
        break
    else:
        initial_guess = next_guess

```

Which returns this:

```

exit at loop 3 with result 1.4142135623730951

```

It took only 3 iterations to achieve a level of accuracy that was better than 1 part in 1 million.

Whilst the accuracy and speed is good, it is noted that this method works well for well behaved functions. If the derivative is close to zero, then the next guess (the Newton step) will be far away and the method will be slow. So in summary this method, whilst powerful, should be used selectively.

We will not go into this further aside from noting that python offers good root solving in the scipy module.

```

from scipy.optimize import fsolve

```

```
def f(x): return x**3-100*x**2-x+100
fsolve(f, [2,90])

# returns array([ 1., 100.]
```

## Graph a square

An exercise for the user.

$$x^2 + y^2 = 1$$

$$x^4 + y^4 = 1$$

$$x^8 + y^8 = 1$$



## Dynamic systems

The computers of today are much better equipped to model evolving systems that involve many components. We touched upon the idea of a  $n$ -body simulation of galaxy mergers in the animations section. Indeed it would be very nice to observe the gravitational pull between two colliding galaxies consisting of thousands of stars.

This concept actually maps over into nearly every aspect of science, such as fluid dynamics, disease dynamics, the habits of crowd behaviour, small and large biological systems such as trees and forests and so forth. Each one is interesting in its own right and we can learn lessons from both the mathematics and the computer models.

## Infectious diseases

Infectious diseases is a topical subject in 2022 as we follow the aftermath of a flu type of virus which was determined to be global.

So let's model and examine how a disease might manifest itself. The first thing to note is that the dynamics will be sensitive to the rules and initial conditions that we set. By this we mean that small changes in the rules or initial conditions might have outsized proportionate effects in the results. Indeed, it is these very small changes that one might accidentally or otherwise invoke, that could lead to radical outcomes that lead to a mass hysteria in the general population, which is in itself another dynamic system that can be modelled.

Noting the above, the model created here will be similar, but yet different to other models that are created.

Here are some initial assumptions:

1. At the start  $n\%$  of the population is sick.
2. A person can move in a defined grid area.
3. Two adjacent people can transmit the disease.
4. The disease persists in a person for  $t$  days.
5. A person can be sick, but will survive.

For the purpose of keeping this model concise, we will stick to the basic rules. But remember that we can easily change the parameters or even think of and add more criteria or remove other criteria that we deem to be unsuitable.

The outcomes from all models, including this one, will always have the same features.

1. Benign: nothing happens.
2. Mass spreading then dissipates.
3. Steady state: persistent existence.

Let's define a person class:

```
import numpy as np
```

```

class Person:

    def __init__(self, id=0, pct_sick=0.01, maxgrid=100):
        self.id = id
        self.sick = int((np.random.random()<pct_sick))
        self.maxgrid = maxgrid

        self.immune = False
        self.xpos = np.random.randint(1,self.maxgrid,1)[0]
        self.ypos = np.random.randint(1,self.maxgrid,1)[0]
        self.age = np.random.randint(1,90,1)[0]
        return

```

We then add a method to the class for moving the person and also making sure that the person exists within the boundary:

```

def movePerson(self, step_size):
    # Generate a random integer number [1,2,3,4]
    pos = np.random.randint(1,5,1)[0]

    if pos == 1:
        self.ypos += step_size
    elif pos == 2:
        self.ypos -= step_size
    elif pos == 3:
        self.xpos += step_size
    elif pos == 4:
        self.xpos -= step_size

    # Check boundaries
    if self.xpos < 0:
        self.xpos = 1
    if self.xpos > self.maxgrid:
        self.xpos =self.maxgrid

    if self.ypos < 0:

```

```

        self.ypos = 1
    if self.ypos > self.maxgrid:
        self.ypos = self.maxgrid

    return

```

And lastly a method for maintaining the health of a person. This is where we might set the number of days that a person can carry the virus and also add immunity (although for example, we know that immunity could fade away over time).

```

def updateHealth(self, boardXY):
    # Neighborhood
    neighxv = np.array([self.xpos, self.xpos+1, self.xpos+1, \
                        self.xpos+1, self.xpos, self.xpos-1, \
                        self.xpos-1, self.xpos-1])
    neighyv = np.array([self.ypos+1, self.ypos+1, self.ypos, \
                        self.ypos-1, self.ypos-1, self.ypos-1, \
                        self.ypos, self.ypos+1])

    neighxv[neighxv > self.maxgrid] = self.maxgrid
    neighxv[neighxv < 0] = 0
    neighyv[neighyv > self.maxgrid] = self.maxgrid
    neighyv[neighyv < 0] = 0

    neighv = np.zeros(8)
    for i,(x,y) in enumerate(zip(neighxv,neighyv)):
        neighv[i] = boardXY[x,y]

    if self.sick > 0:
        self.sick += 1

    if self.sick > 14:
        self.sick = 0 # recovered and healthy again
        self.immune = True

    # If you bump into a sick person-field, make guy sick
    if ((self.sick == 0) and (np.sum(neighv) > 0))

```

```

    and (self.immune == False)):
        self.sick = 1 # newly sick

    return

```

Now that a ‘person class’ which gives the basic rules is set up, we need to make functions that move the people after each round of the simulation and also we should start by making a function that can plot charts:

```

def f_plotPeople(nrPeople, persList):
    """ make interim plots after each simulation round.
    One simulation round represents a day in our setup
    """
    fig = plt.figure()
    ax = plt.subplot(111, aspect='equal')
    ax.set_title('Healthy vs Sick Population')

    xposSickv = np.array([], dtype=int)
    yposSickv = np.array([], dtype=int)
    xposHealthyv = np.array([], dtype=int)
    yposHealthyv = np.array([], dtype=int)
    for i in range(nrPeople):
        person = persList[i]
        if person.sick > 0:
            xposSickv = np.append(xposSickv, person.xpos)
            yposSickv = np.append(yposSickv, person.ypos)
        else:
            xposHealthyv = np.append(xposHealthyv, person.xpos)
            yposHealthyv = np.append(yposHealthyv, person.ypos)

    ax.plot(xposSickv, yposSickv, marker='s', linestyle = 'None', \
            color = 'red', markersize=5, alpha=0.6)
    ax.plot(xposHealthyv, yposHealthyv, marker='s', linestyle = 'None', \
            color = 'green', markersize=5, alpha=0.6)
    plt.legend(['Infected', 'Healthy'], bbox_to_anchor=(1, 0.75))
    plt.show()

```

```
return
```

Now we create a function for updating the community's (list of people's) health and positions.

```
def f_update(nrPeople, persList:list[Person], boardXYin, xmax, ymax, step_size):  
    """  
    updates the population of people by first allowing  
    the person to move according to the step_size variable.  
    If step_size equals zero the person self isolates and  
    does not move at all.  
    """  
    nrSick = 0  
    for i in range(nrPeople):  
        person = persList[i]  
        person.movePerson(step_size)  
        person.updateHealth(boardXYin)  
  
    # Make new board and mark all the sick fields  
    boardXY = np.zeros([xmax+1, ymax+1])  
    for i in range(nrPeople):  
        person = persList[i]  
        if person.sick > 0:  
            nrSick += 1  
            boardXY[person.xpos, person.ypos] = 1  
  
    return nrSick, boardXY
```

And lastly a simulation routine which updates the board if a person is sick

```
def f_simulateCase(nrPeople,step_size,xmax,ymax,maxIter,i_plotBoard):  
    persList = []  
  
    boardXY = np.zeros([xmax+1, ymax+1])  
    nrSickv = np.zeros(maxIter, dtype=int)
```

```

# Make person list
for i in range(nrPeople):
    # Make person
    person_i = Person(id = i, pct_sick=pct_sick, max_grid=xmax)

    # If person is sick update board
    if person_i.sick == True:
        boardXY[person_i.xpos, person_i.ypos] = 1

    # Store person in list
    persList.append(person_i)

for i_loop in range(maxIter):
    nrSickv[i_loop], boardXY = \
        f_update(nrPeople, persList, boardXY, xmax, ymax, step_size)

    if i_loop <10 or i_loop>(maxIter-5):
        print('-----')
        print(f'Round {i_loop} num sick {nrSickv[i_loop]}')

    if ((i_loop%10 == 0) and (i_plotBoard == 1)):
        f_plotPeople(nrPeople, persList)

return nrSickv

```

At this point we are now ready to run the simulation. All we need to do is set the parameters. This model is relatively simplistic, so all we need is are:

- The area (maxgrid)
- The number of people
- The percentage of sick people to start with
- And a step\_size (how far a person can travel)

```

# Set random seed
mySeedNumber = 143895784

# Maximum days to simulate
maxIter = 70

```

```

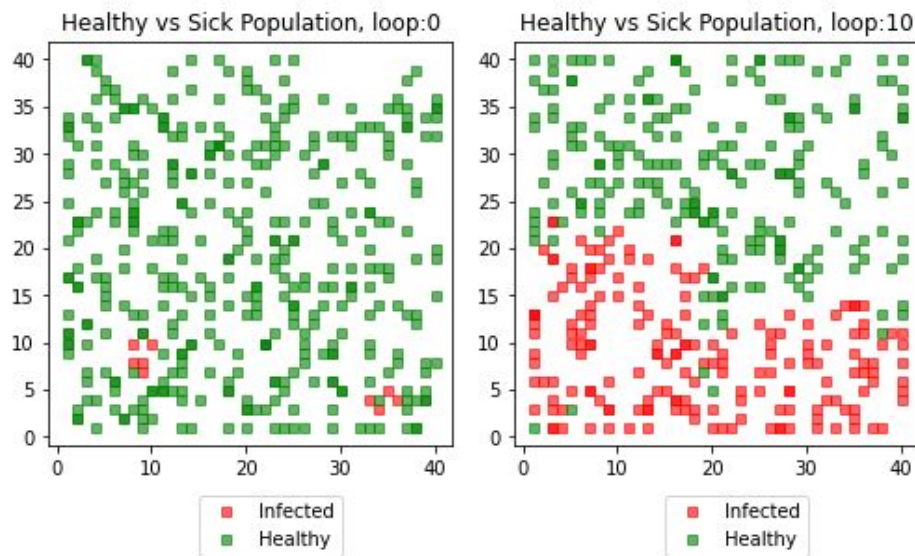
# Size of their world
maxgrid = 40
xmax = ymax = maxgrid

# People
pct_sick = 1/100
nrPeople = 400

# Case 1: Move a lot (no restrictions)
np.random.seed(mySeedNumber)
i_plotBoard = 1
step_size = 2 # How far they travel (interaction radius)
nrSick1v = f_simulateCase(nrPeople,step_size,xmax,ymax,maxIter,i_plotBoard)

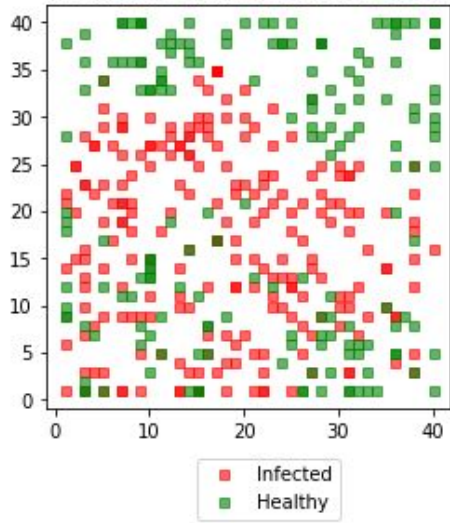
```

With these conditions we get the following output plots (representing a snapshot of every 10 days in the cycle):

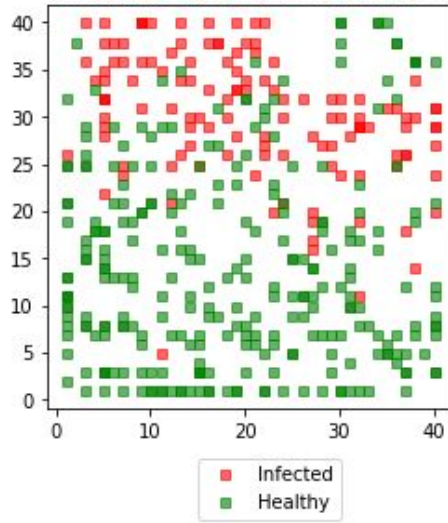




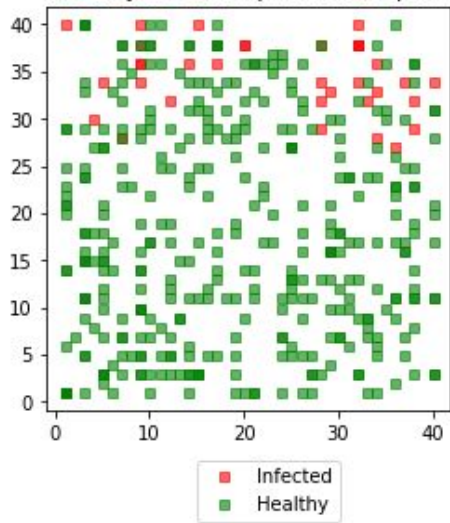
Healthy vs Sick Population, loop:20



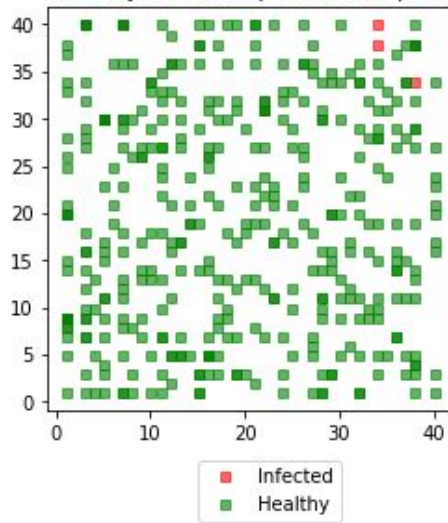
Healthy vs Sick Population, loop:30

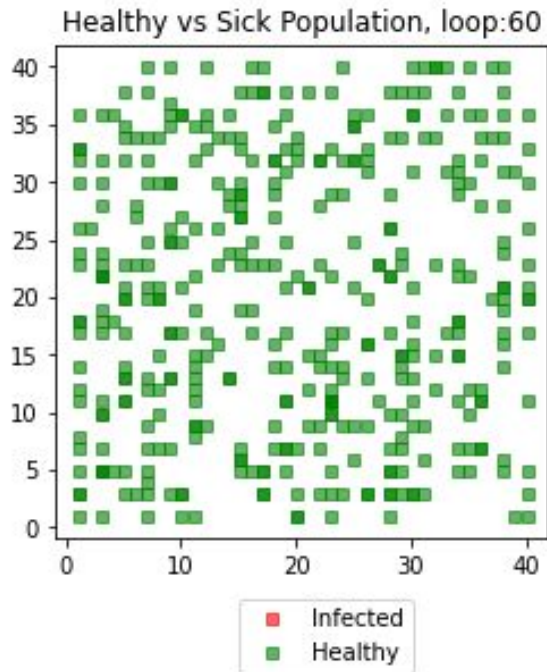


Healthy vs Sick Population, loop:40

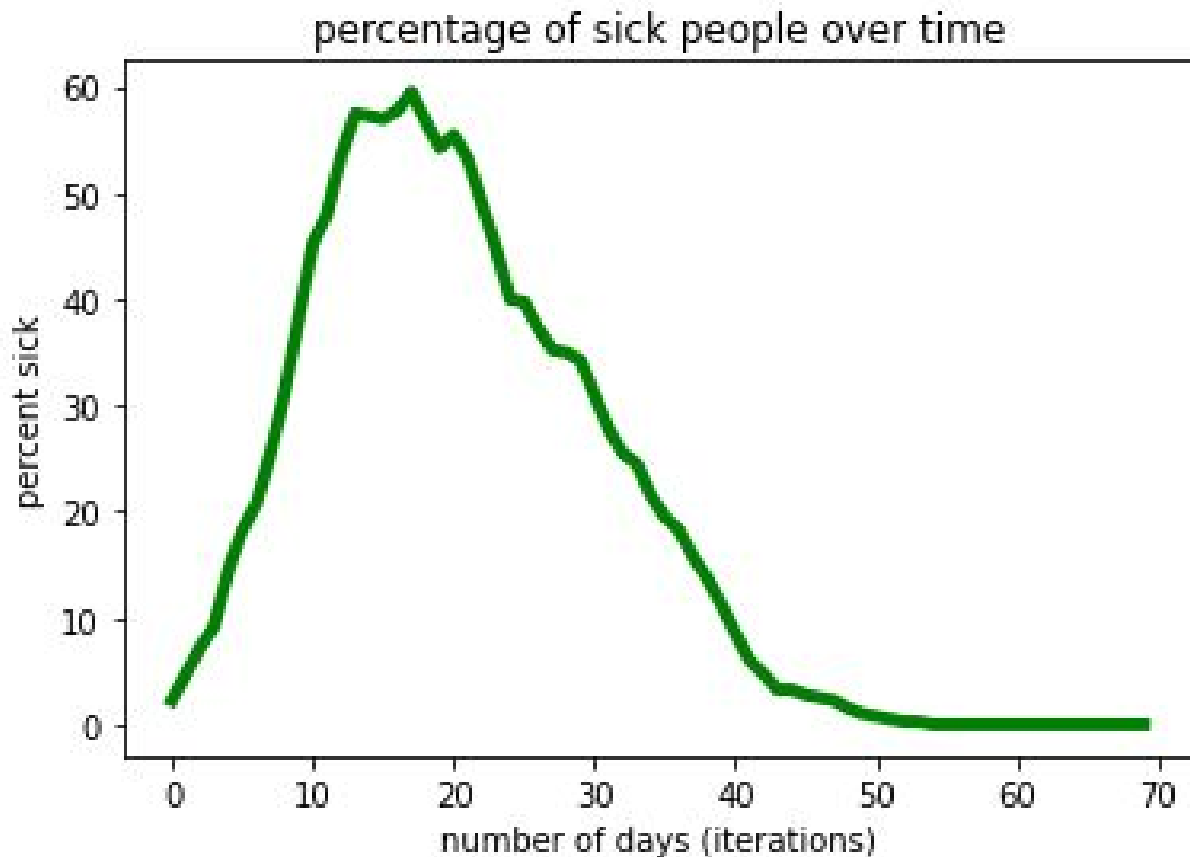


Healthy vs Sick Population, loop:50



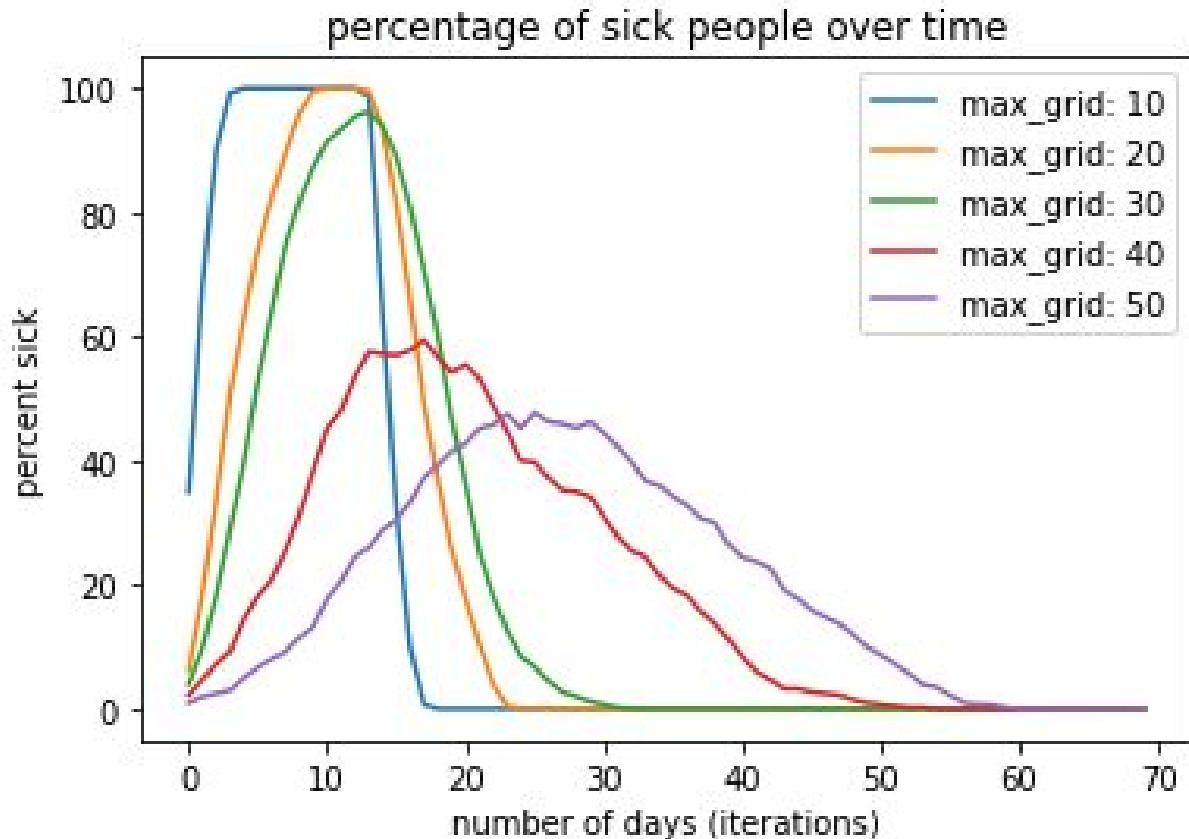


Even with the most basic assumptions it is clear to see that the number of sick people quickly increases from 1% of the population to well over 50% by the 20th day before rapidly declining to zero by day 60.



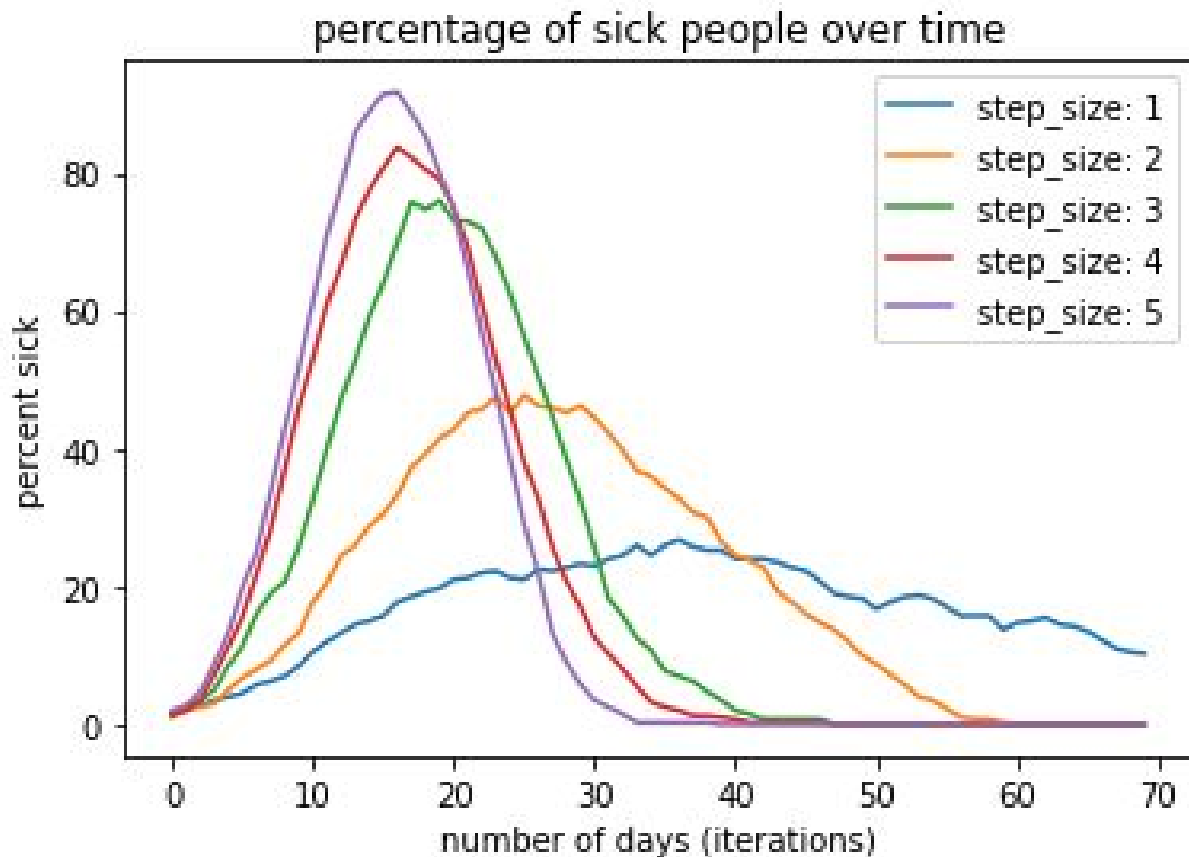
Factors that affect the rate of growth will be population density (the number of people per unit area) and internal movements within the community. If the model included a function that killed a portion of the sick people, then the disease would fade away quicker too as those people would no longer be classified as spreaders.

We can illustrate this by increasing the population density (by keeping the same number of people but changing the area), which would result in a family of curves that look like this:



If the density is high (small grid area), then the percentage of sick people jumps to a high percentage quickly and fades away fast and the converse is true when the density is low.

Likewise, we can do the same type of simulation but this time varying the step size to get this:



Again, we see that more movement increases the percentage of sick people, but also shortens the life of the virus.

In many ways we might expect the above features in the same way that water boils in a pot. As the temperature rises, the internal movement of the particles increases. This increases the collisions and therefore the transmission of heat.

In summary, the model shows that we can change the temporal profile of the virus penetration throughout the population, but ultimately, in order to gain immunity, all people will need to have contracted the virus in the first instance.

## Rumour dynamics

We looked at infectious diseases and how they spread. A virus was carried from one person to the next by some kind of closest neighbour mechanism.

The model showed that lower population densities and smaller movements reduces the propagation although ultimately the same number of people would catch the virus at some point as ultimately immunity needs to be achieved on the individual basis.

So now we transfer our focus on the mathematics of how a rumour might propagate through a system. If we consider the rumour itself as a transfer of information irrespective as to if it is a truth ( True ) or a lie ( False ). For example, the rumour could have been about the severity of the virus in the previous section (scaremongering<sup>[23]</sup>) or some other kind of truth or lie about a completely unrelated subject.

So what are the rules of the transfer of a rumour?

1. A certain percentage of people start the rumour.
2. Any person that hears the rumour tells their friends.
3. The person no longer tells others and stops caring.

Broadly speaking, with the above rules and our knowledge of the model results from infectious disease, we might intuitively guess that the propagation dynamics will in some way be shared.

Our rumour model does not have a physical proximity characteristic because the various forms of social media mean that the transmission does not require physical interaction. However, *the same proximity characteristic will manifest itself in the sense that a person tells their closest friends.*

The above translates into the mathematics of the connectivity of people. A *more connected system* could be perceived to be analogous to a densely populated system with lots of movement (boiling water in a pot). Ultimately, the transmission is expected to be high in this scenario.

Knowing that the transmission is high, we now have to be aware of well connected people (super spreaders). A well connected person (such as a social influencer or a media channel) will propagate the message faster.

A person who is a super spreader transmits in the same way as a person who is highly infectious and moves around a lot in the infectious disease model.

So let's try to set up a model but again respecting many of the inputs that we could include but have elected not to for the purpose of simplicity.

We start with some initial conditions and set up a person class:

```
import random
import matplotlib.pyplot as plt

# community of size n
num_people = 100_000
community = [x for x in range(num_people)]
community_noifications = []

class Person:

    def __init__(self, id=0) -> None:
        """ base conditions of a person to start """
        self.id = id # unique id
        self.friends = self.generate_friends()
        self.motivation = 90 # motivation to transmit info
        self.aware = False # is the person aware
        self.care = False # if aware, does the person care
        self.notified = 0 # counter for times notified
        self.message_sent = 0 # counter for messages sent
        return
```

These are the attributes that we want, but there are a whole myriad of options and ways that we can model the rumour.

Examples of parameters that we have elected to omit or bundle into one parameter.

- Spreader through media
- Spreader through verbal communication
- Communication channel

The parameters here are suitable. In particular:

1. **Self.aware**: is the person aware of the rumour?

2. **Self.care**: does the person care to spread the rumour?
3. **Self.notified**: how many times has the person been notified. If a person gets too many notifications, then they lose interest.
4. **Self.message\_sent**: if a person has sent too many messages, they get bored and stop.

Next we define some methods for the class: Firstly we generate a list of friends.

```
def generate_friends(self) -> list:
    """ generate a list of friends from the community """
    friends = random.choices(community, k=5)
    friends = list(set(friends))
    return friends
```

We have made a small list for the purpose of the example, but even with this small number, the saturation point is reached after only several iterations.

Next, some rules for updating a person who has received a notification:

```
def update_person(self):
    """ update the persons status """
    self.notified += 1
    if self.notified > 1:
        self.motivation = self.motivation - 10

    if self.aware == False:
        # just heard a rumor, now spread it.
        self.aware = True
        self.care = True

    if self.notified > 3:
        # now bored so dont care
        self.care = False
```

We add logical rules, such as motivation declining and people getting disinterested in the rumour.

And finally, a method for sending messages to friends.

```
def message_friends(self, community_noifications:list):
```



```

    """ message friends making them aware """

    if self.aware == True and self.care == True:
        self.message_sent += 1
        self.motivation = self.motivation - 10
        if self.message_sent > 10:
            # sent enough messages, now bored
            self.care = False

    for i in self.friends:
        if self.motivation > random.randint(0,100):
            community_noifications.append(i)

```

We use motivation in a probabilistic way. Each person's motivation level declines which means it is less likely that they will forward on a rumour to friends in their friends list.

We can now start with the body of the code:

```

# generate a list of people
person = [Person(x) for x in community]

# a few people start a rumour
x = random.choices(community, k=3)
for i in x:
    person[i].aware = True
    person[i].care = True
    person[i].notified = 1

```

We use list comprehension to set up all of the people in the community in one powerful line. This is a specific feature of python that other languages would need to express verbosely. And from the community, the model selects three people at random to start the rumour.

Lastly, we can generate the simulation by looping through the process:

```

# do some loops
who_is_aware = 0
who_cares = 0

```

```

community_aware = []
community_care = []
community_motivation = []

for i in range(40):
    # each aware person notifies friends
    for j in person:
        j.message_friends(community_noifications)

    # the people receive notifications
    for j in community_noifications:
        person[j].update_person()

    # updates complete, so empty the list
    community_noifications = []

    # collect the data from each loop
    who_is_aware = 0
    who_cares = 0
    motivation_level = 0
    for i in person:
        if i.aware == True:
            who_is_aware += 1
        if i.care == True:
            who_cares += 1
        motivation_level = motivation_level + i.motivation/100
    community_aware.append(who_is_aware)
    community_care.append(who_cares)
    community_motivation.append(motivation_level)

```

We create lists so that we can plot the data later. The loop itself contains a cycle of messaging friends and then other friends receiving those messages.

Finally, we can plot the data:

```

plt.plot(community_motivation, label='community motivation')
plt.plot(community_aware, label='community aware')
plt.plot(community_care, label='community care', linewidth=4)

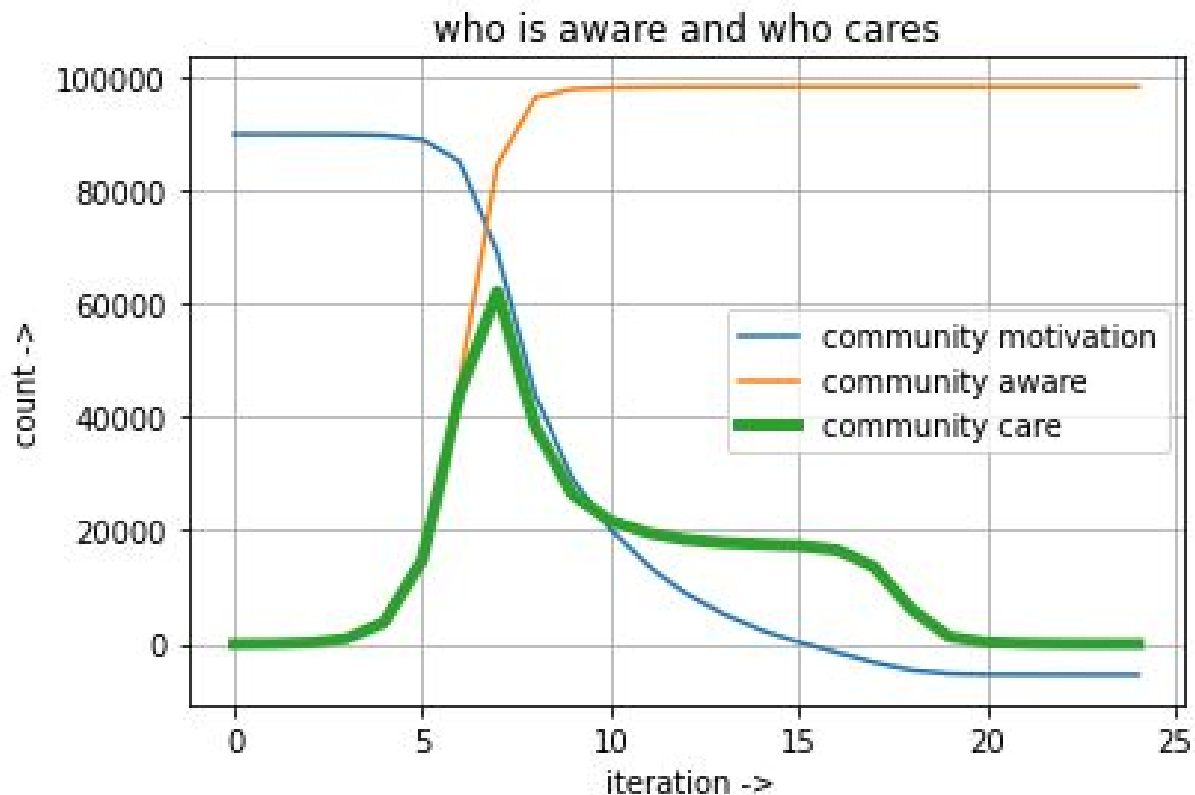
```

```
plt.title('who is aware and who cares')
plt.xlabel(' iteration ->')
plt.ylabel('count ->')
plt.legend()
plt.grid()
plt.show()
```

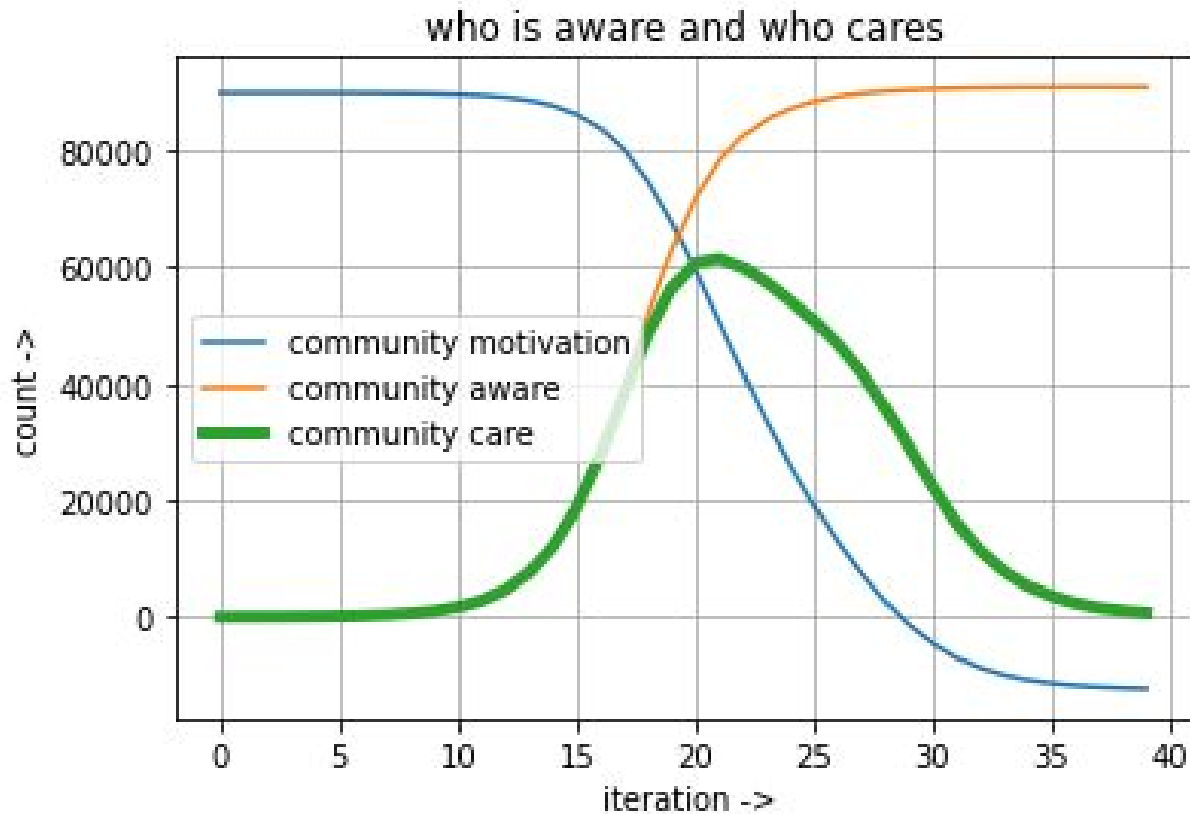
The three lines we care for are:

1. Community awareness: Here we find a typical s-shaped curve where there is a steep incline and then saturation.
2. Community care: This is the number of people at any point in time that care to send on the rumour. Here we see a point of “peak” interest, before people get bored of the rumour.
3. Community motivation: is an alternative measure for community care. It represents how motivated in total the community is to transmit the rumour. We see that this waynes with time.

Our first plot is the standard model.



Our second plot is similar to the first, but with the motivation levels reduced. The reduction of motivation smooths and extends the curve as it takes longer to reach a saturation point.



In fact, we see that the maximum saturation level is lower when the motivation levels are reduced. This is because the rumour has died out before it has reached the entire population (ie. people have gotten too bored too quickly).

The entire contents of the code was just 120 lines in python and yet produced extremely powerful computations and results.

## Fractals

A Fractal is a curve or geometrical figure, each part of which has the same statistical character as the whole. They are useful in modelling structures in which similar patterns recur at progressively smaller scales, and in describing partly random or chaotic phenomena.

Fractals could be said to be part of the dynamic systems. Indeed, if we look at the world around us, it can quite easily be seen that we are surrounded by hundreds of fractals often without even knowing it.

Let's create a fractal using the basic turtle module:

```
import turtle as ttl

ttl.shape('turtle')
ttl.speed(0)

def tree(size, levels, angle):
    if levels==0:
        ttl.color('green')
        ttl.dot(size)
        ttl.color('black')
        return

    ttl.forward(size)

    ttl.right(angle)
    tree(size*0.8, levels-1, angle)

    ttl.left(angle * 2)
    tree(size*0.8, levels-1, angle)

    ttl.right(angle)
    ttl.backward(size)

ttl.left(90)
tree(70, 9, 60)
```

```
ttl.mainloop()
```

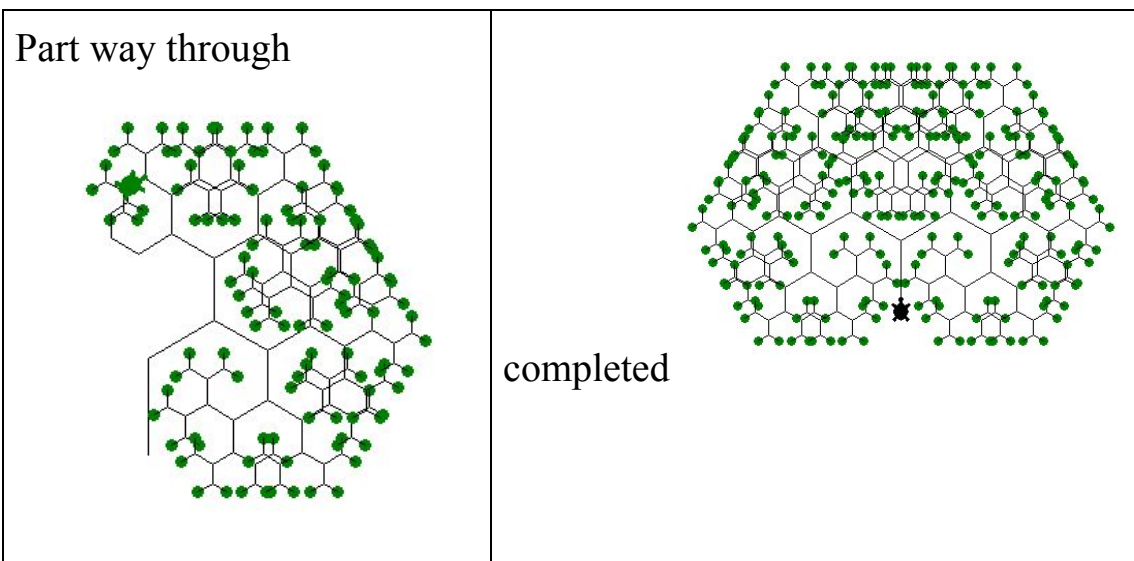
One of the properties of fractals is the repeated nature and this fits well computationally with recursive functions.

We touched on recursive functions much earlier and found two convenient formats. [1] for the function to repeatedly call itself (simple), [2] calling a function repeatedly in a loop (efficient). In both cases we needed to make sure that we reach a case case when the function ends otherwise we run into an infinite loop or get a stack-overflow.

In the code example above, we create a tree by using a recursive function to repeatedly call itself for the left branches and the right branches.

The base case is set by the if statement where we check if `levels = 0` and then draw a leaf (end node) and return without calling the function any more.

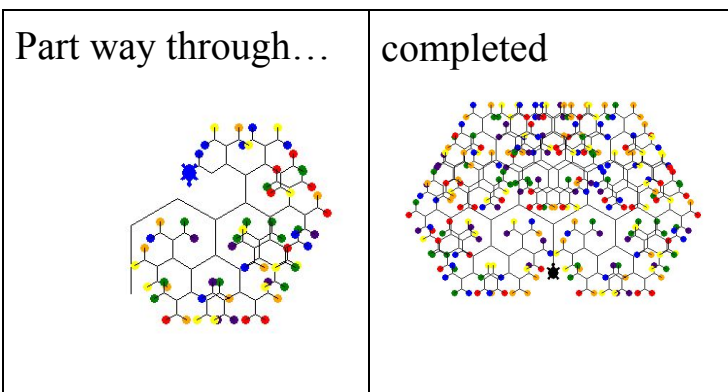
The turtle sets about passing through all of the recursive paths until every node is reached. This is the result of the code:



We can modify the code to add randomness at certain parts, for example, setting random angles or changing the colours of the end nodes. Randomising the colours is done simple by adding the following two lines of code:

```
colour_choice = ['red','orange','yellow','green','blue','indigo']  
ttl.color(random.choice(colour_choice))
```

This achieves a tree with rainbow coloured leaves as per the diagram below.

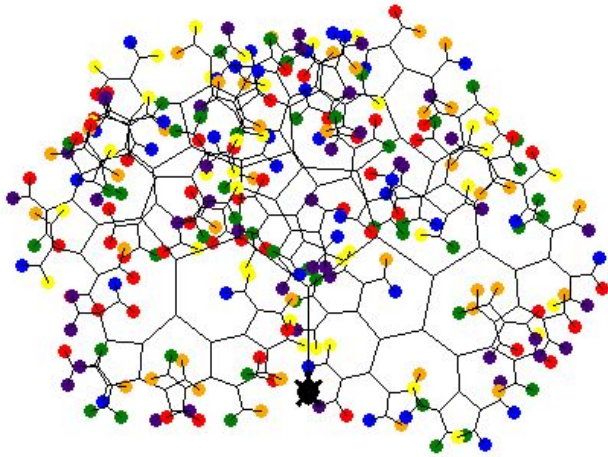


We can let the angle vary randomly within a predefined constraint which would achieve a non-symmetric tree that closely resembles the original tree, but not quite.

We do this by setting:

```
angle = random.randint(30, 90)
```

With this line of code, we remove control over where the turtle will go within the constraints of 30 to 90 degrees and the outcome now looks like this:



The reality is that nature contains both of the above. Ordered fractals (like a regular lattice) and unordered fractals (like the coastline of a country).

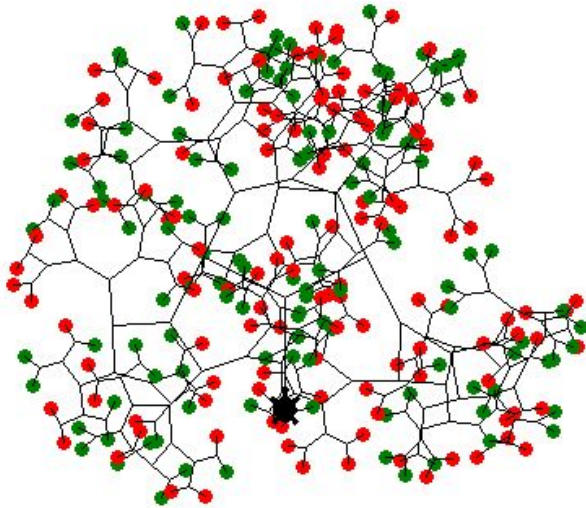
Interestingly, even though there was some form of chaos, the turtle in the code managed to find its way back to the original spot because of the recursive nature of the code.

We can increase the chaos by widening the boundary conditions from [45, 75] degrees to [30,90] degrees and we can also change the code such that we colour the wide angled nodes red and the tighter angled nodes green.

```
if angle > 60:  
    angle_colour = 'red'  
else:  
    angle_colour = 'green'
```

This highlights to the viewer the path taken by the turtle.





But even with this simple model, we can do better in terms of replicating the ordered chaos of nature. Let's say for example that the tree has a tendency to move in one direction, towards the sunny side, we can include this into the basic model by changing the probabilities.

In doing so, we will create more leaves of one colour than another by roughly the same proportion as the bias that we have introduced.

Let's create a skewed list function. For this we can use the scipy module with the method `skewnorm.rvs()` .

```
from scipy.stats import skewnorm
import matplotlib.pyplot as plt

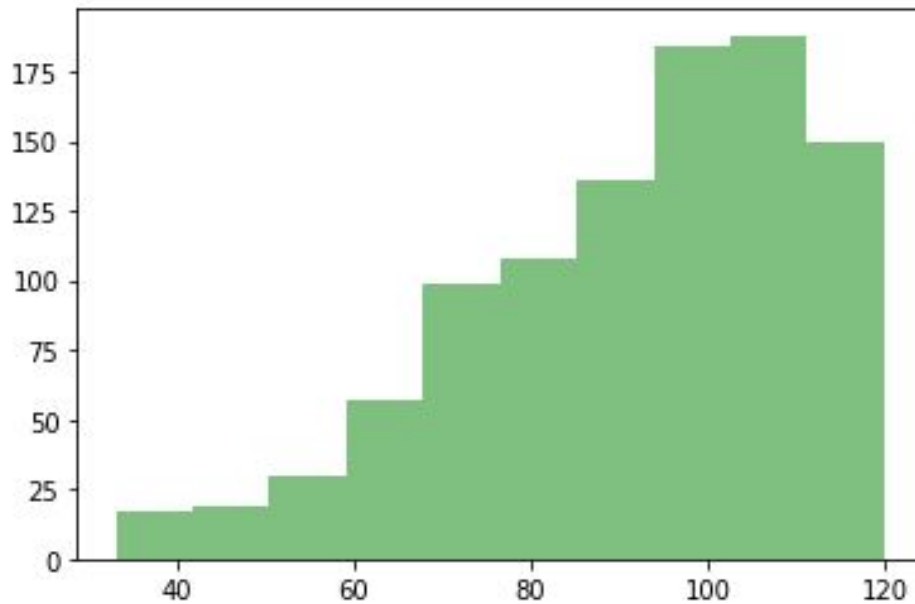
def skewed_list(skew, size=1000, x_low=30, x_high=120):
    """ returns a skewed list between x_low and x_high """
    data= skewnorm.rvs(skew, size=size)*100
    data = data - min(data)
    data = data / max(data)
    data = data * x_high
    data = [x for x in data if x > x_low]

    return data

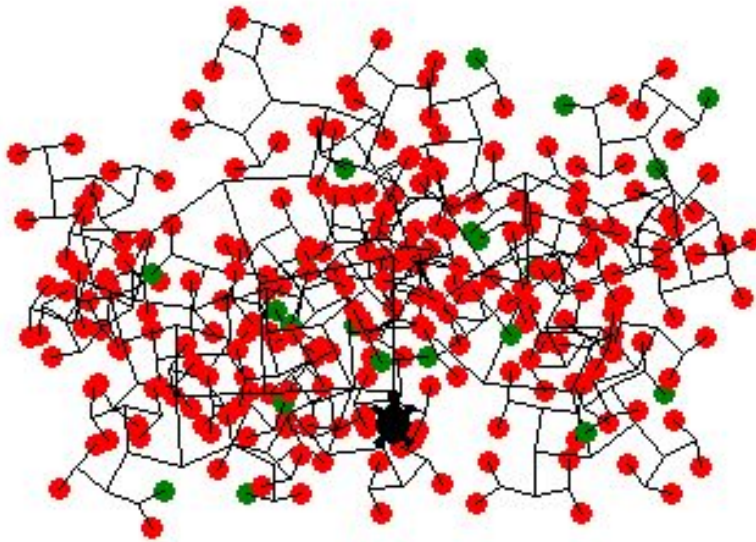
r = skewed_list(-30, size=1000, x_low=30, x_high=120)
```

```
plt.hist(r, color='green', alpha=0.5)  
print(min(r))
```

The list produces a distribution that generically looks like this:

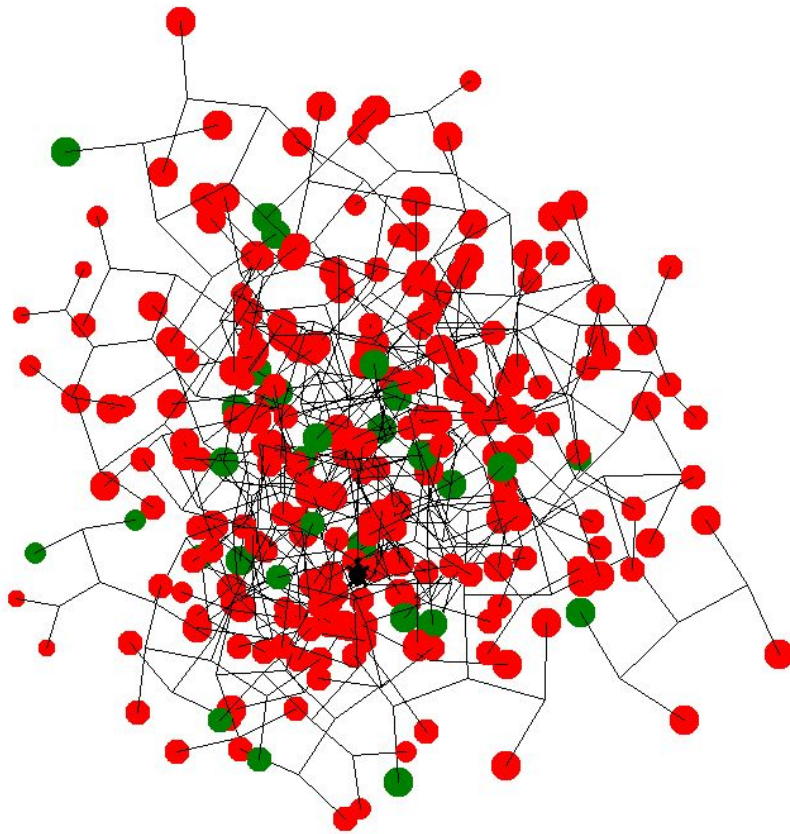


We use the unevenly distributed list for the turtle's angle changes and therefore its colours will be unevenly distributed (according to the skewed distribution) too.

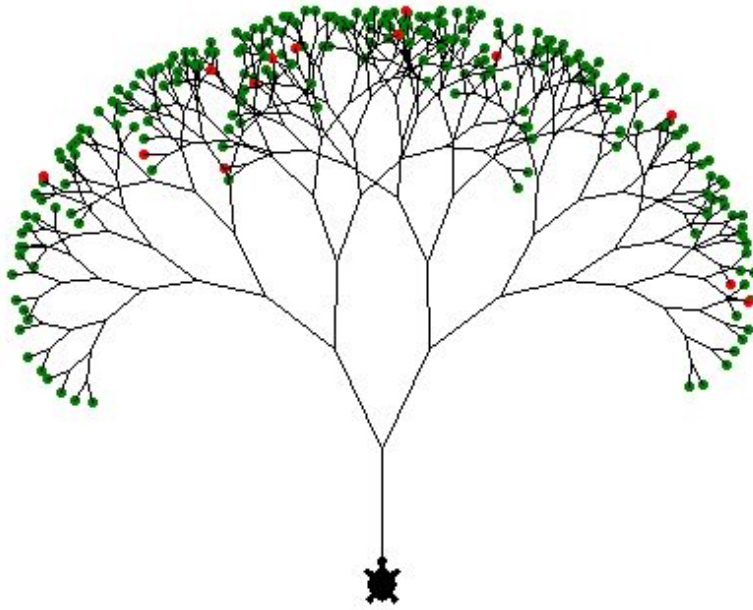


What we observe is a random network of mostly red leafs with a generally wider angle between the branches.

Again, finally, changing the size (which represents the length of each branch) does this:



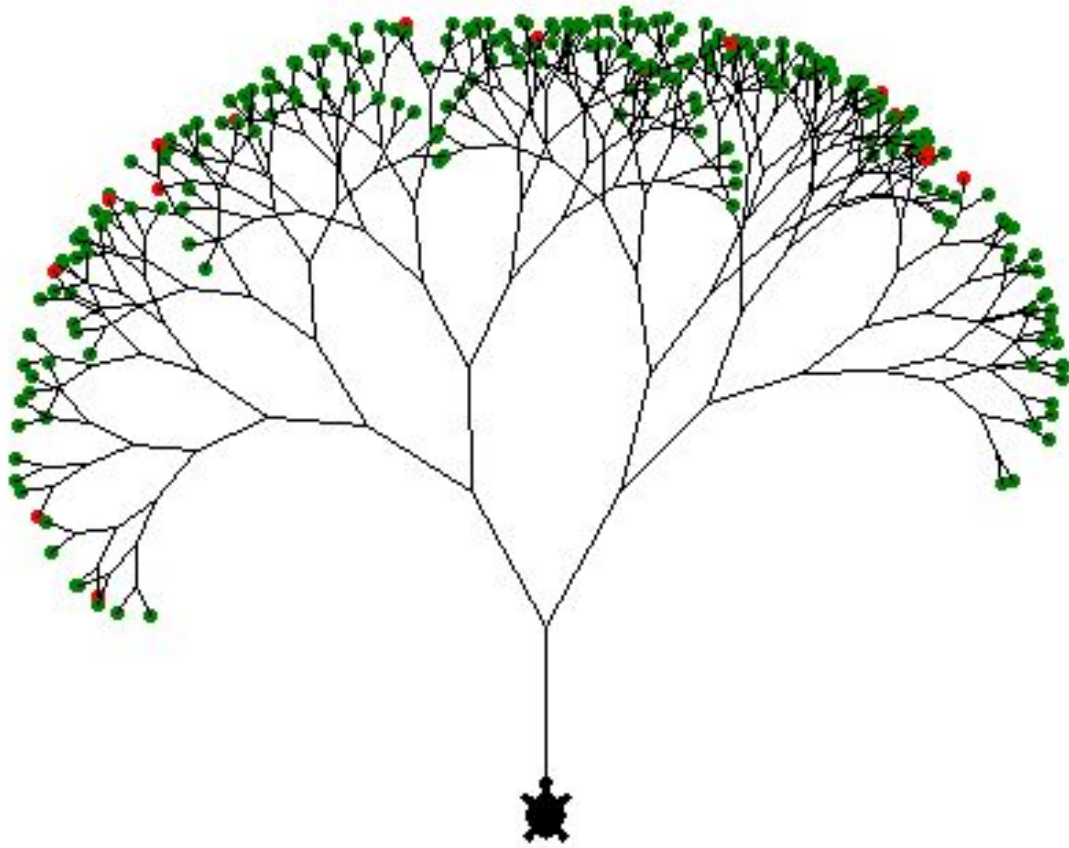
And finally, let's plot a cherry tree using angles ranging between 10 and 30 degrees in the final model. This constrains the tree to be upward looking just as regular trees in nature are. We also switch the colouring such that the majority of the leaves, based on the uneven probability distribution, are green. So the resulting picture is this:



In every tree, by use of the recursive function, we were always able to return the turtle to the original place, no matter what level of complexity was used either for the branches or the angles.

This code of no more than 60 lines was able to provide the fundamentals for a whole variety of tree type fractals with the user only needing to change a few parameters or criteria.

We can also notice that order exists even though each branch and path was completely random in both length and angle. The nodes end up in a beautiful semi-circle like pattern almost as if by design and if we were to keep on redrawing the trees we would get similar, but not identical patterns, just like human fingerprints are similar, but not identical.

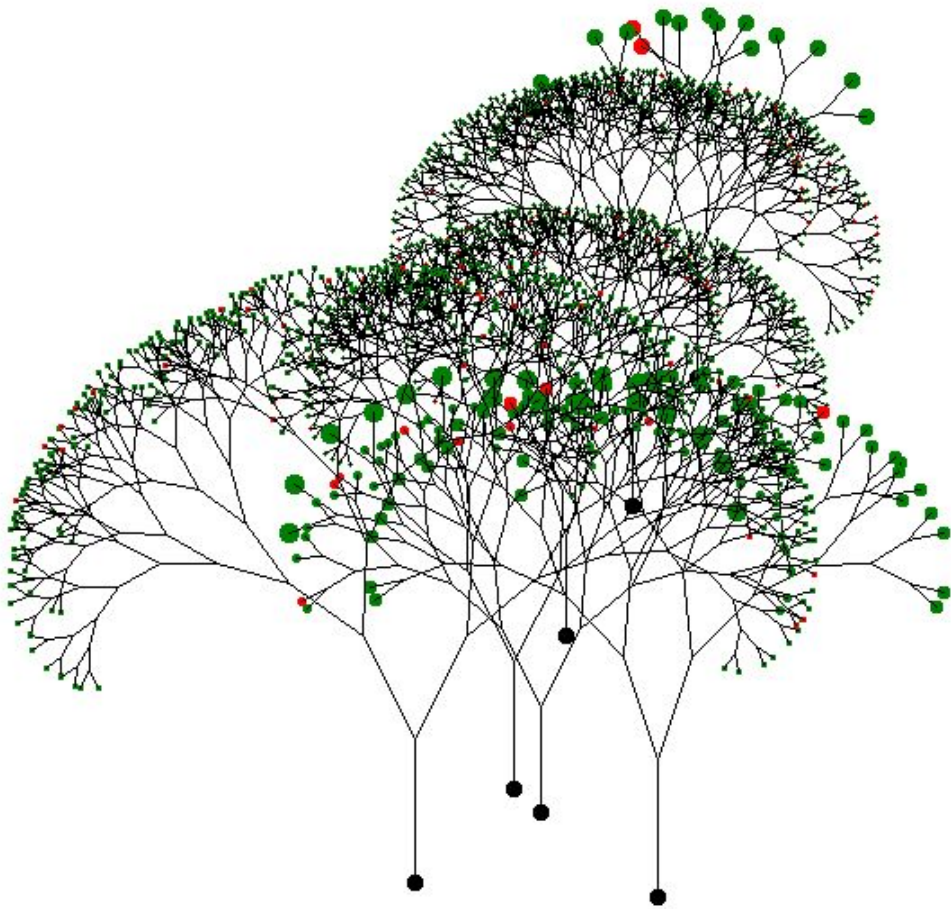


In fact, just like we grow single trees in a loop, we could construct a forest of trees in the same way, with a short adjustment to the code by moving it into a for loop.

```
for i in range(7):  
    # draw several trees  
    ttl.left(90)  
    tree_size = random.choice(skewed_list(-30, x_low=50, x_high=80))  
    tree_levels = random.choice([5,6,7,8,9,10])  
    tree(tree_size, tree_levels)  
    # move to new random location  
    ttl.right(90)  
    ttl.penup()  
    n = random.randint(0,360)  
    ttl.right(n)
```



```
t1.forward(random.randint(1,5)*10 + 60)  
t1.left(n)  
t1.pendown()
```



## Mandelbrot set

Interesting because they combine complex numbers with fractals and also, we learn about stability, which is actually true for all iterative processes. The stability is beautifully highlighted in the mandelbrot set.

Interestingly, this highly complicated pattern is produced by the most simple formula:

$$Z_{n+1} = Z_n^2 + c$$

Where  $Z_n$  are complex numbers.

The way that the mandelbrot set changes is also the hallmark of chaos theory. In the finalised version beautiful patterns are produced and the colouring is often used to represent the stability of any given number (the likelihood of a number blowing up) and this rapid change in stability near the edges is chaotic.

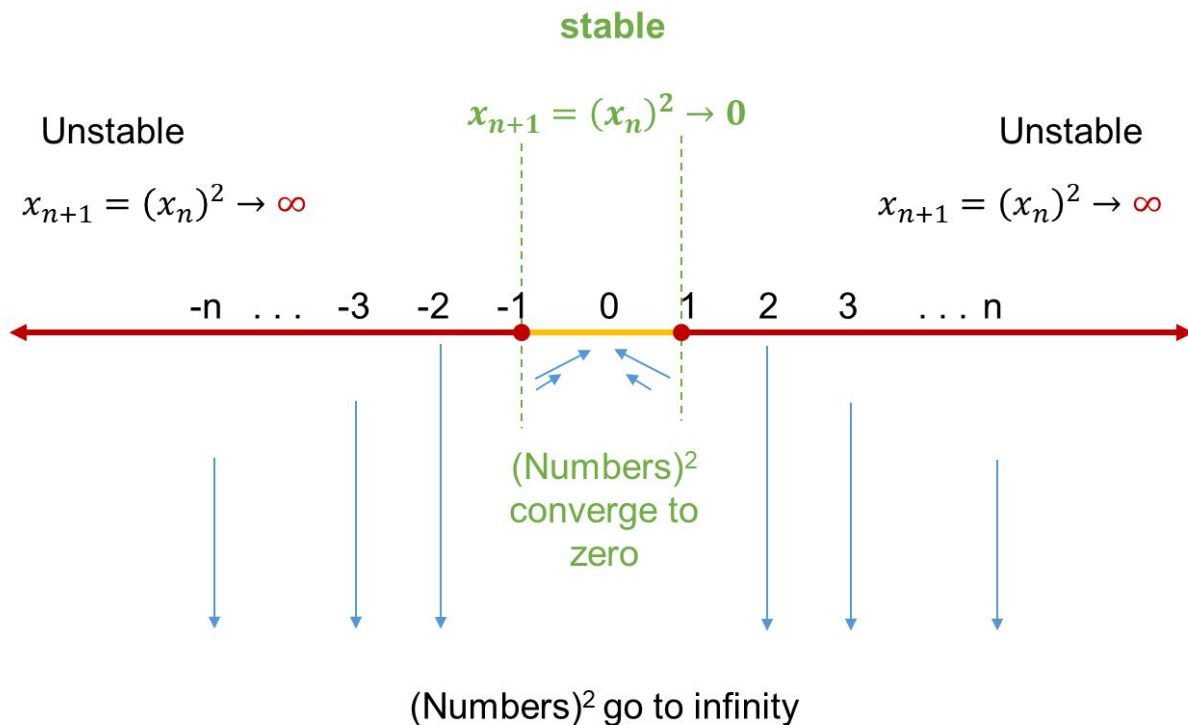
The mandelbrot set originated in the 1970's and by the 1980's the first visualisations were done. However, the computing power was nowhere near the capabilities of today, so the creators were restricted to the mathematics and what computing resource that they had at the time. So this is what they produced:



Before we consider complex numbers, let's think about what happens to regular numbers on the number line and their stability. So let's look at  $X_{n+1} = X_n^2 + c$  where  $X$  are real numbers.

[illegible]

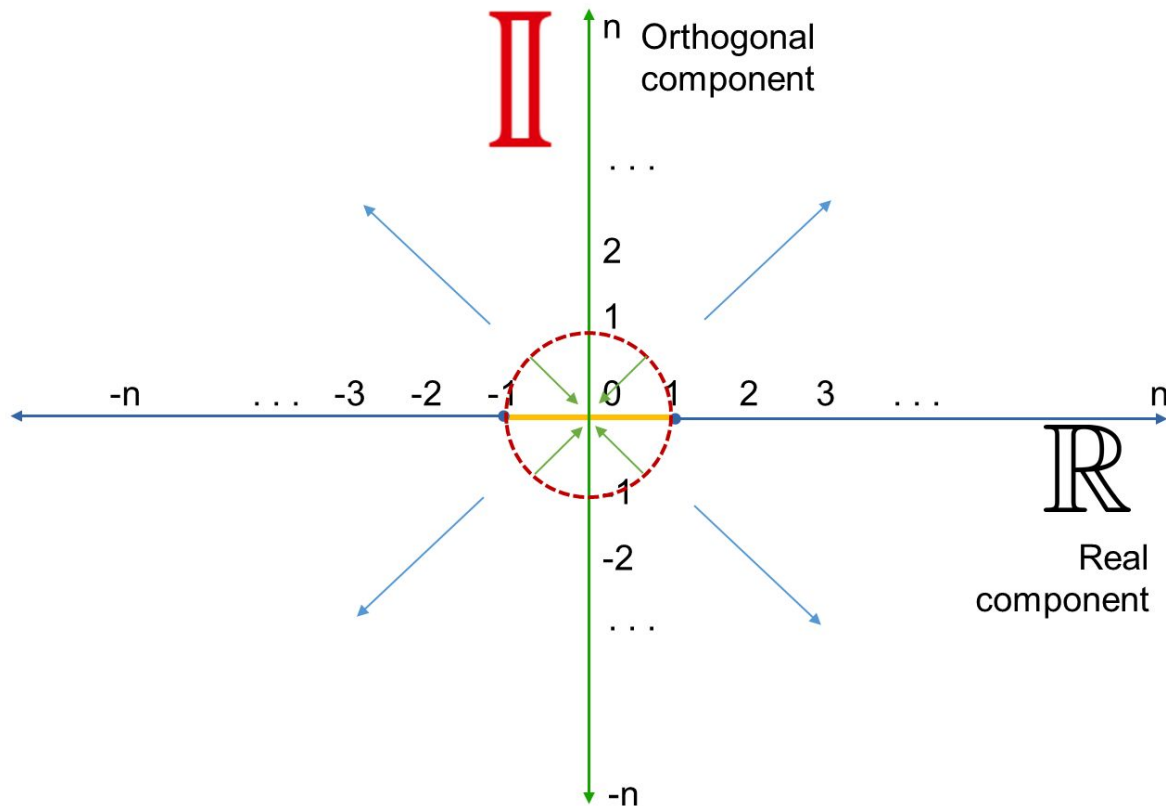
The convergence zone (or radius of convergence) for real numbers is best described by the number line below.



We use this analogy to extend into the complex plane. Extending  $x_n$  to  $z_n$  means that  $x_n^2$  becomes  $z_n^2$ . And we know that  $z_n = a_n + i*b_n$  where  $a$  and  $b$  are real numbers.

When we translate this to polar coordinates (as we did in the complex numbers section), this translates into a circle with radius  $\sqrt{a_n^2 + b_n^2}$ .

So the zone of convergence becomes a radius on convergence and looks like this:



The above is simplistic, but highlights in part what we might expect to happen for  $z_{n+1} = z_n^2$ . Extending the same concept to the mandelbrot equation by adding a constant  $c$  produced something that was unexpected, but when explored partly intuitive.

So, let's get down to the code:

We have used the pygame module, but note that we could have used tkinter (or even turtle) to display the image.

```
import pygame
import ctypes
import time
```

```

# to use pygame's functionality.
pygame.init()

# screen dimensions
user32 = ctypes.windll.user32
width = user32.GetSystemMetrics(0)
height = user32.GetSystemMetrics(1)
width, height = int(width/1.1), int(height/1.1)
win = (width, height)
screen = pygame.display.set_mode(win)

# set the pygame window name
pygame.display.set_caption('Mandelbrot Program')

xaxis = width/2 + 100
yaxis = height/2
scale = 200 # Mandelbrot size
iterations = 20 # the more iterations you make the better resolution you'll get

```

Initially, we have just set the main screen parameters as a function of our actual screen size (so that the code works on any size screen). The y-axis is half of the screen height because the image is the same both above and below the axis.

We define a main function which will do two things:

1. The iteration of the loops
2. Allow the user some basic control

```

def main():
    """ main code for Mandelbrot """
    the_iteration()
    the_loop()

```

We then set about doing the main iteration. This is where the formula  $z_{n+1} = z_n^2 + c$  is invoked.

[illegible]

```

textRect_scale = text_scale.get_rect()
textRect_scale.topleft = (width//10, height//10 + 30)
screen.blit(text_scale, textRect_scale)

pygame.display.update()

```

Most of the code around the main iteration is for human visualisation, such as the colour scales and the text boxes with user info, so there is only one essential line:

```

z = z**2+c # Mandelbrot equation

```

The colour part of the loop is necessary in the sense that it describes to us, the stability of the sequence at each point. Black points are the most stable and are found with this line:

```

if abs(z) > 2:

```

And all the other colours are generated in between this depending on *how many iterations* it takes for the points to become stable.

The last part of the code allows the user to adjust the number of iterations and also the scale.

```

def the_loop():
    global iterations
    global scale

    run = True
    while run:
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                run = False
        keys = pygame.key.get_pressed()
        if keys[pygame.K_LEFT]:
            time.sleep(0.1)
            print('iterations before:', iterations)
            iterations -= 5

```

```

    if iterations < 1:
        iterations = 2
    the_iteration()
    print('iterations after:', iterations)

    if keys[pygame.K_RIGHT]:
        time.sleep(0.1)
        print('iterations before:', iterations)
        iterations += 5
        the_iteration()
        print('iterations after:', iterations)

    if keys[pygame.K_UP]:
        time.sleep(0.1)
        print('scale before', scale)
        scale += 50
        the_iteration()
        print('scale after', scale)

    if keys[pygame.K_DOWN]:
        time.sleep(0.1)
        print('scale before', scale)
        scale -= 50
        the_iteration()
        print('scale after', scale)

pygame.quit()

```

Every time a button is pressed a calculation is done and the screen is redrawn via `pygame.display.update()` . We are already aware of this redrawing of the screen feature in other GUI modules.

And finally to run all the modules (via the main module) we use the common hoisting method:

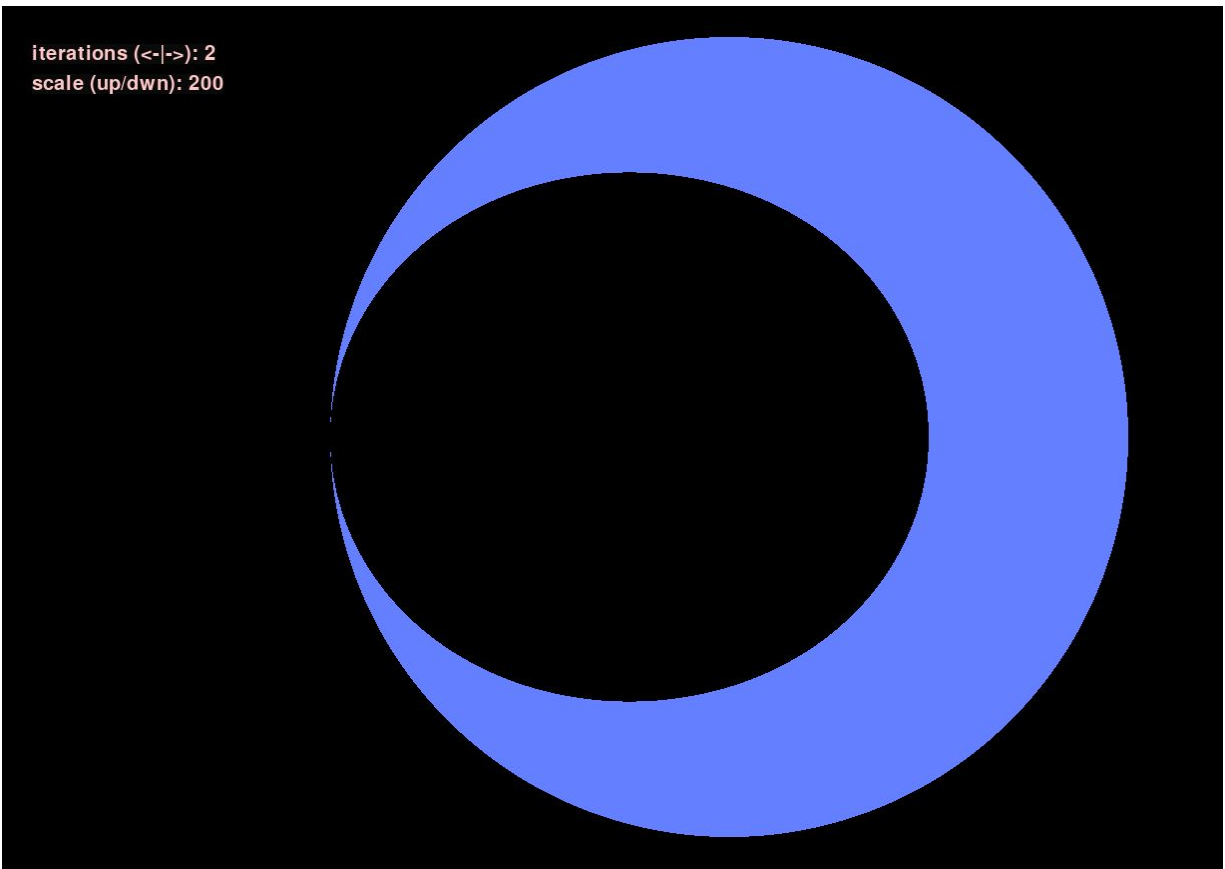
```

if __name__ == "__main__":
    main()

```

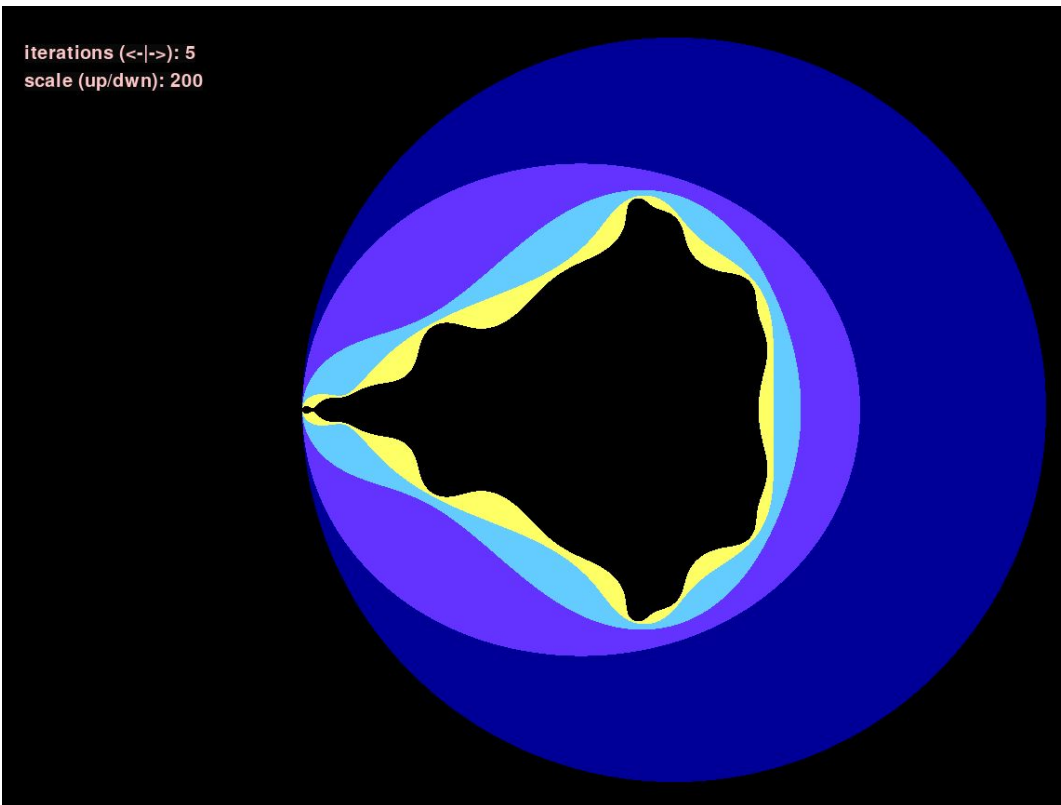
We can now run the code to generate the mandelbrot set with a given number of iterations.

If we use a low number of iterations, then the result is rather boring. Here we have used just 2 iterations:

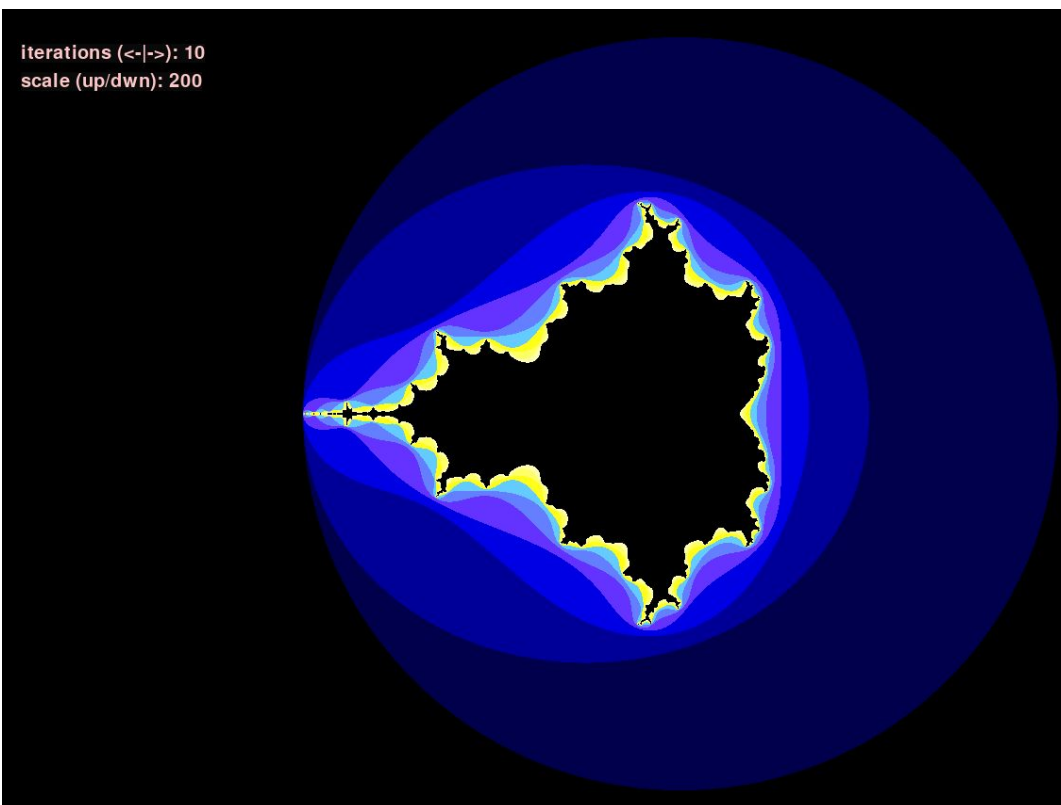


But as we increase the number of iterations, we start to see what appears to be a mundane inner circle, transformed into a different shape with inner features and also the colouring begins to change, which represents the stability of  $z_{n+1}$ .

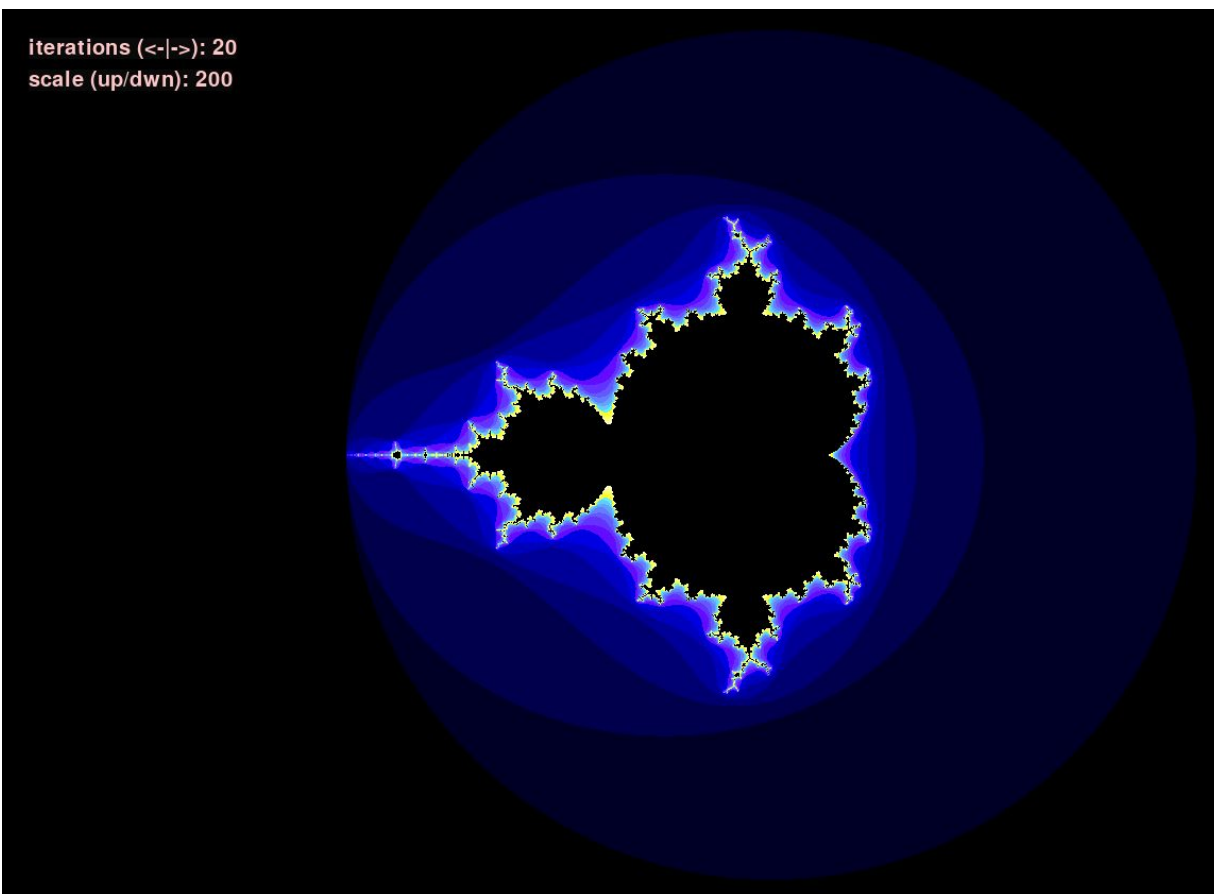




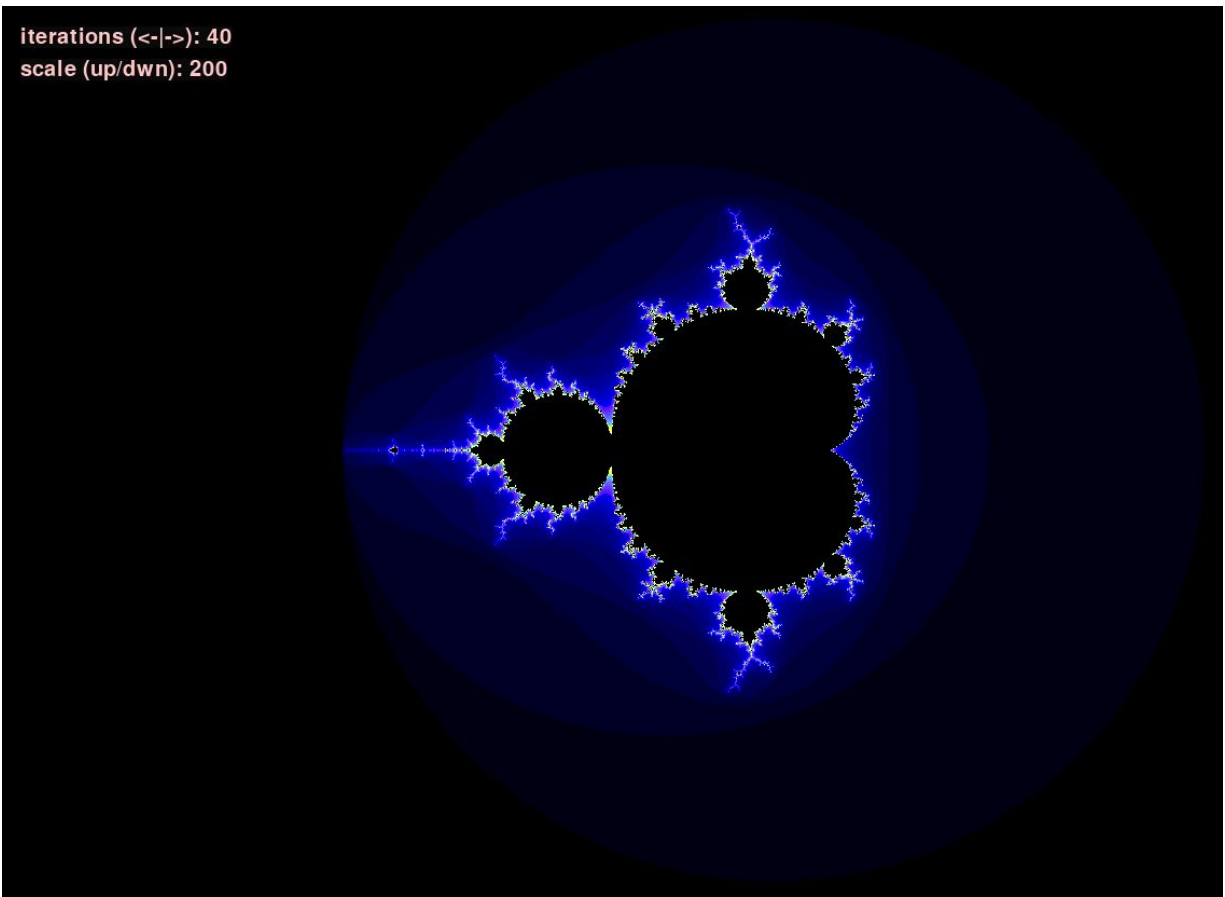
And as the iterations increase more we get this:



And we can keep on going:



All of a sudden at just 20 iterations we begin to see a lot of structure around the edges of the shape.



And this keeps on going indefinitely.

A note to the calculation:. On a standard laptop, there are typically  $1920 * 1080$  pixels so we are looking at the order of 2 million pixels to be computed prior to the loops required. So if we do 20 loops, this takes the process to 40 million calculations.

There are various tricks to reduce the calculation volume and the first one is to use a smaller proportion of the screen as the number of pixels is proportional to the square of the sides.

The research into the mandelbrot set is vast and would take weeks to cover the topic so we have reached a natural stopping point, but what is interesting is the mathematics around the edges, because we find that we can zoom into these points indefinitely and they keep on producing new fractals in a rather chaotic and unpredictable way.

## Double pendulum

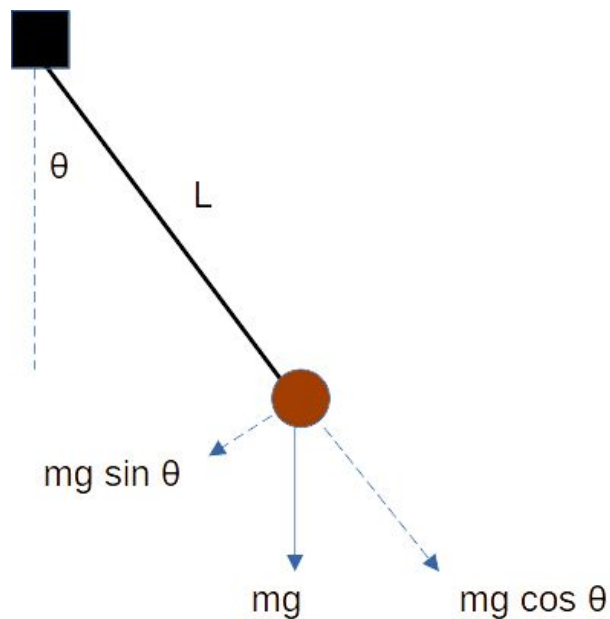
The double pendulum comes directly from the concept of the single pendulum.

The single pendulum is known to be an excellent timekeeper due to the fact that the time taken for a given swing from end to end (a tick-tock in a clock) is constant whatever amplitude the single pendulum is started at. In fact we are able to prove from first principles that the constant time taken,  $T$ , follows this formula.

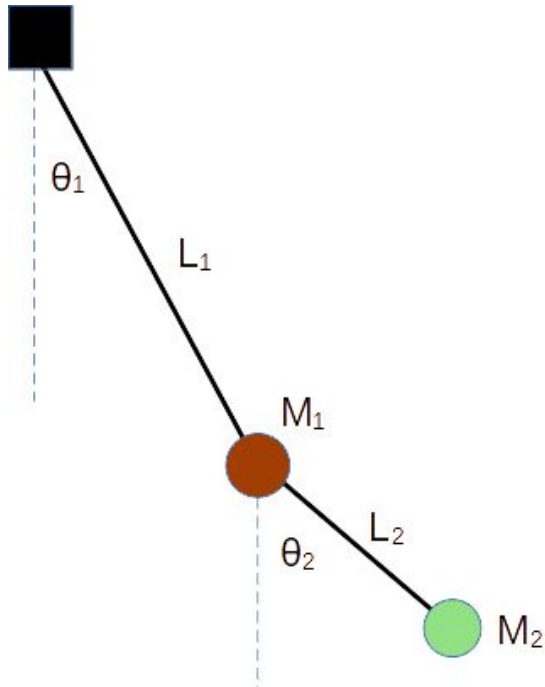
$$T = 2\pi \sqrt{\frac{L}{g}}$$

Where  $L$  is the length of the pendulum and  $g$  is the force of gravity (which is constant on the surface of the Earth). We see that the only physical factor that we can control that determines the period  $T$  is the length of the pendulum  $L$ .

We can do all of the above knowing the basic forces that act upon the pendulum and Newton's laws of motion.



However, the double pendulum is a classic Physics problem that gives rise to an example of chaotic motion where one might have otherwise expected order. In particular, because the setup of the double pendulum looks rather similar to its sibling.



The mathematics behind the problem are lengthy, but essentially we have positions and velocities for each of the pendulum bobs.

Positions:

- $(x_1, y_1) = (L_1 \sin \theta_1, L_1 \cos \theta_1)$
- $(x_2, y_2) = (L_1 \sin \theta_1 + L_2 \sin \theta_2, -L_2 \cos \theta_1 - L_2 \cos \theta_2)$

Velocities:

- $(\underline{x}_1, \underline{y}_1) = (L_1 \underline{\theta}_1 \cos \theta_1, L_1 \underline{\theta}_1 \sin \theta_1)$
- $(\underline{x}_2, \underline{y}_2) = (L_1 \underline{\theta}_1 \cos \theta_1 + L_2 \underline{\theta}_2 \cos \theta_2, L_1 \underline{\theta}_1 \sin \theta_1 + L_2 \underline{\theta}_2 \sin \theta_2)$
-

Where  $\underline{x} = \frac{dx}{dt}$  and  $\underline{\theta} = \frac{d\theta}{dt}$

From the above we can work out the potential energy and kinetic energy of both pendulum bobs knowing that:

$$ke = \frac{1}{2}mv^2 \text{ and } pe = mgh$$

And we know that energy is conserved, such that:

$ke + pe = \text{Constant}$ , which is the total energy in the system known by Hamiltonian mechanics

However, for the purpose of the double pendulum, one uses Lagrangian mechanics which is  $ke - pe$  and represents the difference in the energies. So we say that:

$$L = ke - pe$$

And this fits into the euler-lagrange equation:

$$\frac{d}{dt} \left( \frac{dL}{dq} \right) = \frac{dL}{dq}$$

Which yields the equation of motion of the system without needing to consider the individual forces in the system.

After some calculus and algebra we are left with two lengthy equations that involve second derivatives. But since we know that the second derivative is a derivative on the first derivative, we can put this into the code.

We start with imports that we intend to use:

```
from numpy import sin, cos
import numpy as np
import matplotlib.pyplot as plt
import scipy.integrate as integrate
import matplotlib.animation as animation
```

and set the initial conditions.

```
G = 9.8 # acceleration due to gravity, in m/s^2
L1 = 1.0 # length of pendulum 1 in m
L2 = 1.0 # length of pendulum 2 in m
M1 = 1.0 # mass of pendulum 1 in kg
M2 = 1.0 # mass of pendulum 2 in kg
```

Next we need a function to deal with the derivatives in a given state.

```
def derivs(state, t):

    dydx = np.zeros_like(state)
```

```

dydx[0] = state[1]

delta = state[2] - state[0]
den1 = (M1+M2) * L1 - M2 * L1 * cos(delta) * cos(delta)
dydx[1] = ((M2 * L1 * state[1] * state[1] * sin(delta) * cos(delta)
            + M2 * G * sin(state[2]) * cos(delta)
            + M2 * L2 * state[3] * state[3] * sin(delta)
            - (M1+M2) * G * sin(state[0]))
            / den1)

dydx[2] = state[3]

den2 = (L2/L1) * den1
dydx[3] = ((- M2 * L2 * state[3] * state[3] * sin(delta) * cos(delta)
            + (M1+M2) * G * sin(state[0]) * cos(delta)
            - (M1+M2) * L1 * state[1] * state[1] * sin(delta)
            - (M1+M2) * G * sin(state[2]))
            / den2)

return dydx

```

We then need to set a range of times for the simulation and also the starting angles and angular velocities.

```

# create a time array from 0..100 sampled at 0.05 second steps
dt = 0.05
t = np.arange(0, 20, dt)
print(t)

# th1 and th2 are the initial angles (degrees)
# w10 and w20 are the initial angular velocities (degrees per second)
th1 = 120.0
w1 = 0.0
th2 = -10.0
w2 = 0.0

# initial state
state = np.radians([th1, w1, th2, w2])

```

We then use the scipy module to integrate the ordinary differential equation that we originally derived.

```
# integrate your ODE using scipy.integrate.  
y = integrate.odeint(derivs, state, t)
```

For each value of t we have one state. So we produce the y values which is a list of lists and looks like this:

```
[[ 2.0943951  0.      -0.17453293  0.      ]  
 [ 2.08189245 -0.49983043 -0.18037987 -0.23120247]  
 [ 2.04447081 -0.99609112 -0.19711468 -0.43005342]  
 ...  
 [-0.25507841  4.38151899 -3.8135224  3.26796836]  
 [-0.04126843  4.14049477 -3.67188173  2.37875988]  
 [ 0.15636407  3.74412399 -3.57692326  1.41412786]]
```

Both the length of t and states are the same.

We have effectively stored, for each t, both of the angles and the angular speeds.

It is now trivial to plot the  $(x_1, y_1)$  and  $(x_2, y_2)$  positions given that we know the angles  $(\theta_1, \theta_2)$  and the lengths  $(L_2, L_2)$  of the pendulums at each point in time.

```
x1 = L1*sin(y[:, 0])  
y1 = -L1*cos(y[:, 0])  
  
x2 = L2*sin(y[:, 2]) + x1  
y2 = -L2*cos(y[:, 2]) + y1
```

So we can now set about plotting the data. In fact, we can actually plot the moving simulation of the double pendulum with this:

```
fig = plt.figure()  
ax = fig.add_subplot(111, autoscale_on=False, xlim=(-2, 2), ylim=(-2, 2))  
ax.set_aspect('equal')  
ax.grid()  
  
line, = ax.plot([], [], 'o-', lw=2)  
time_template = 'time = %.1fs'  
time_text = ax.text(0.05, 0.9, "", transform=ax.transAxes)  
  
def init():
```



```

line.set_data([], [])
time_text.set_text("")
return line, time_text

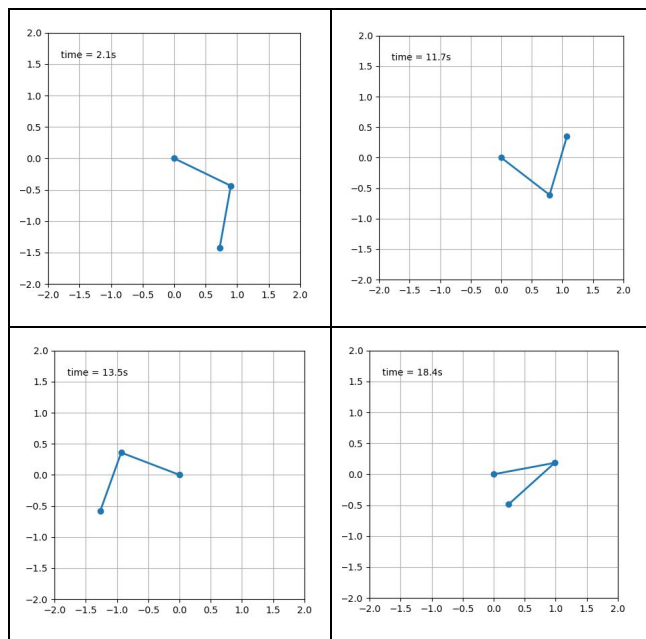
def animate(i):
    thisx = [0, x1[i], x2[i]]
    thisy = [0, y1[i], y2[i]]

    line.set_data(thisx, thisy)
    time_text.set_text(time_template % (i*dt))
    return line, time_text

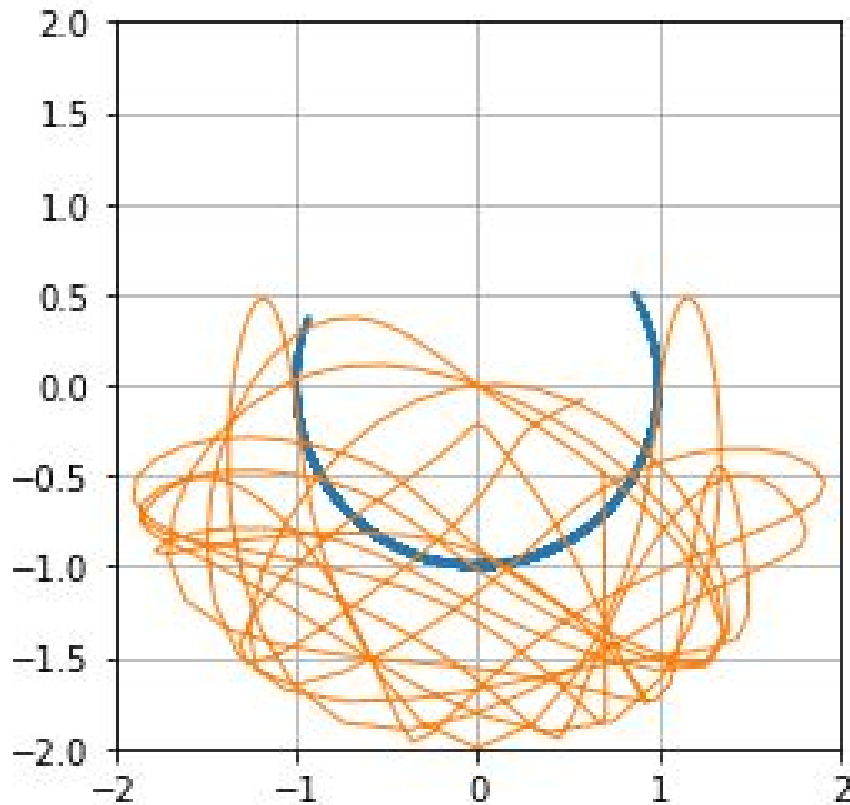
ani = animation.FuncAnimation(fig, animate, range(1, len(y)),
                             interval=dt*1000, blit=True, init_func=init)
plt.show()

```

Which gives this moving animation series:



But for the purpose of this exercise, we can also plot the path taken by the pendulums and in particular, the second pendulum.



We see that the first pendulum is an orderly curve and this must be the case given that there is only 1 degree of freedom  $\theta_1$  and the length is fixed,  $L_1$ .

However, the second pendulum follows an unpredictable and chaotic path. The second pendulum is constrained within certain constraints, for example it cannot reach further than  $|L_1| + |L_2|$  and cannot be shorter than  $|L_1| - |L_2|$ . However, within these constraints, we cannot tell where the pendulum bob will be at any given point in time.

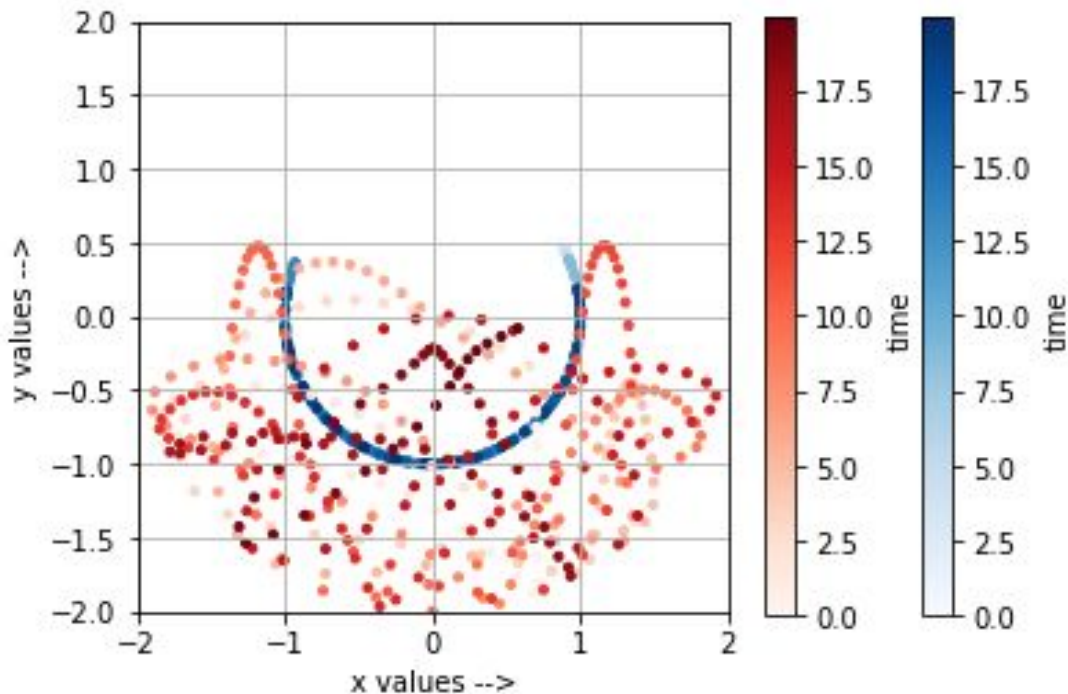
We can show how the pendulums *move with time* by reducing both plots to scatter plots like this:

```
cmap1 = plt.colormaps['Blues']
cmap2 = plt.colormaps["Reds"]

sp1 = ax.scatter(x1, y1, c=t, marker='.', cmap=cmap1)
sp2 = ax.scatter(x2, y2, c=t, marker='.', cmap=cmap2)
ax.set_xlabel('x values -->')
ax.set_ylabel('y values -->')
fig.colorbar(sp1, label='time')
fig.colorbar(sp2, label='time')
```

```
plt.show()
```

Which



The interesting aspect is to note what happens to the path of the second pendulum when any of the parameters is changed by a small amount.

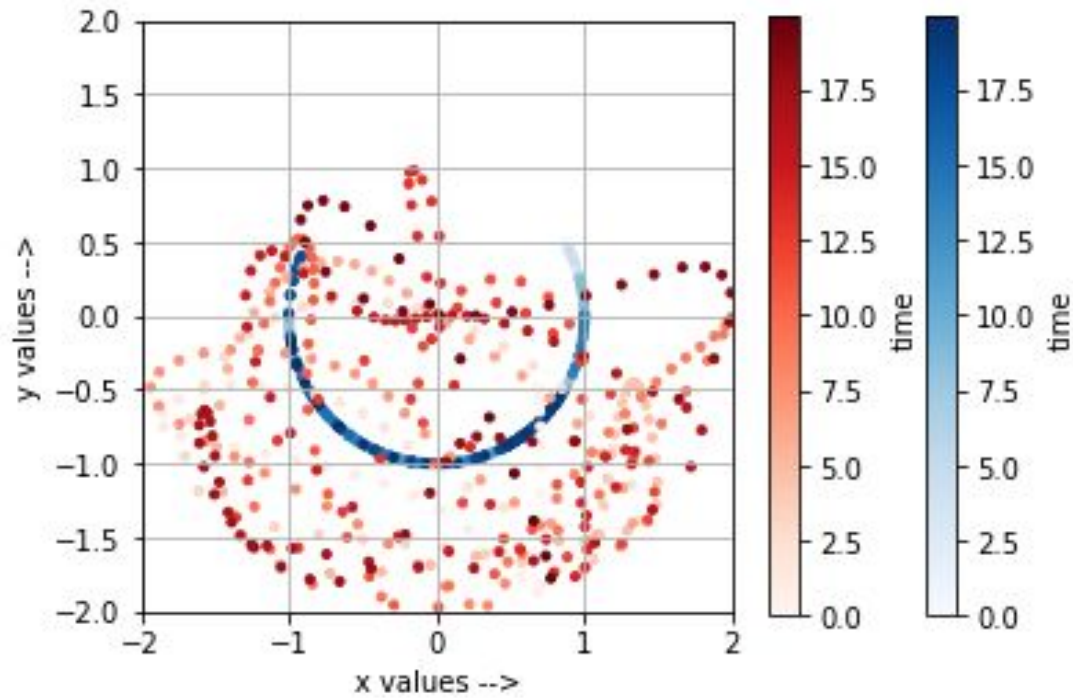
So we set a new initial state (called delta) like this:

```
# initial state
state = np.radians([th1, w1, th2, w2])
state_delta = np.radians([th1, w1+0.01, th2, w2])
```

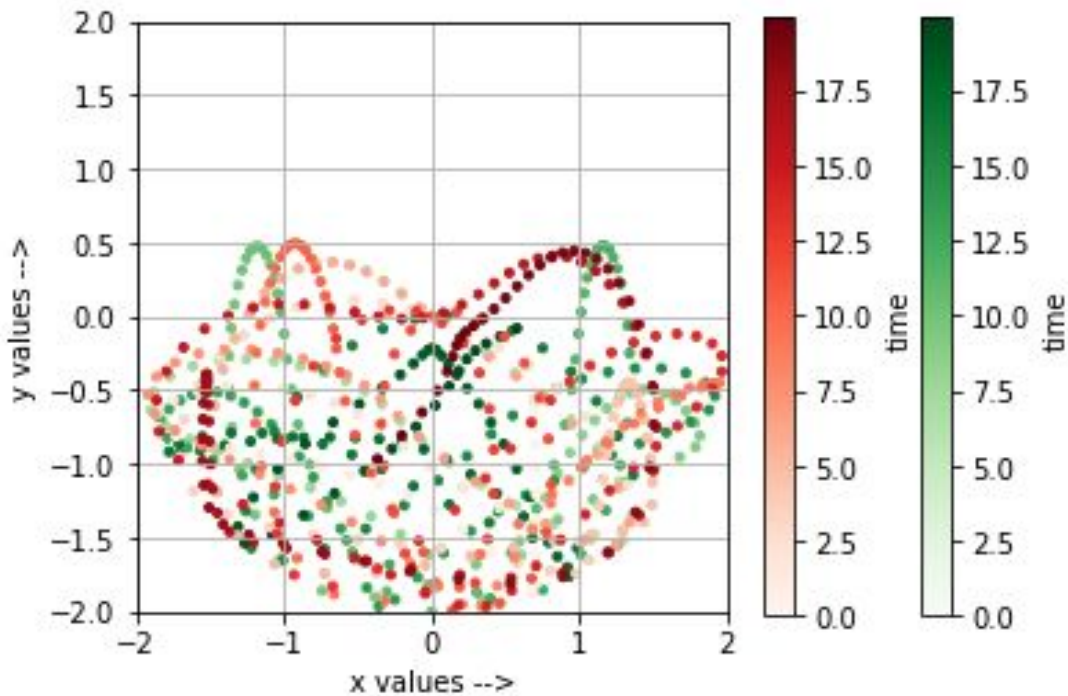
And we can use this to generate new y values like this:

```
# integrate your ODE using scipy.integrate.
y = integrate.odeint(derivs, state, t)
y_delta = integrate.odeint(derivs, state_delta, t)
```

And from this small change ( $w_1 \rightarrow w_1 + 0.01$ ) we generate a completely different plot.



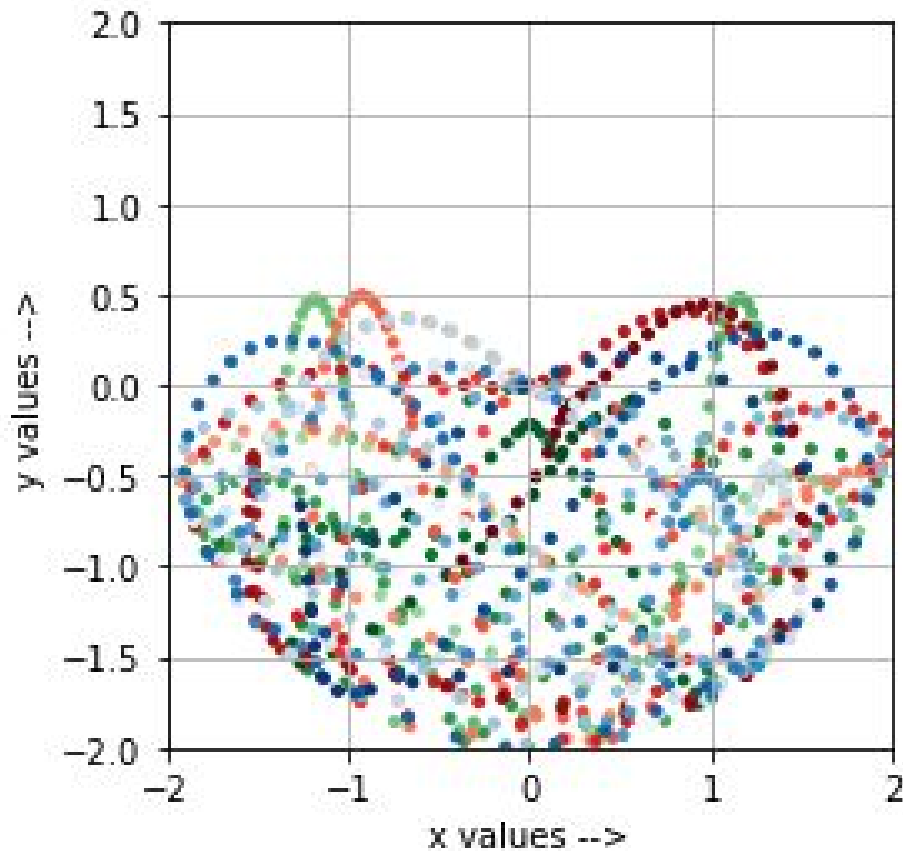
In fact, we can compare the two plots for the second pendulum bobs side by side (we remove the upper pendulum bob for clarity as these will overlap for the same  $L_1$ ).



And we can keep on introducing new plots even with the smallest changes, for example a third state this time by amending  $\theta_1 \rightarrow \theta_1 + 0.01$ .

```
# initial state
state = np.radians([th1, w1, th2, w2])
state_delta = np.radians([th1, w1+0.01, th2, w2])
state_delta_2 = np.radians([th1+0.01, w1, th2, w2])
```

We get a third completely different plot



Other than being retained within certain boundaries, we have shown by computer simulation that the double pendulum is sensitive to initial conditions and chaotic in motion.

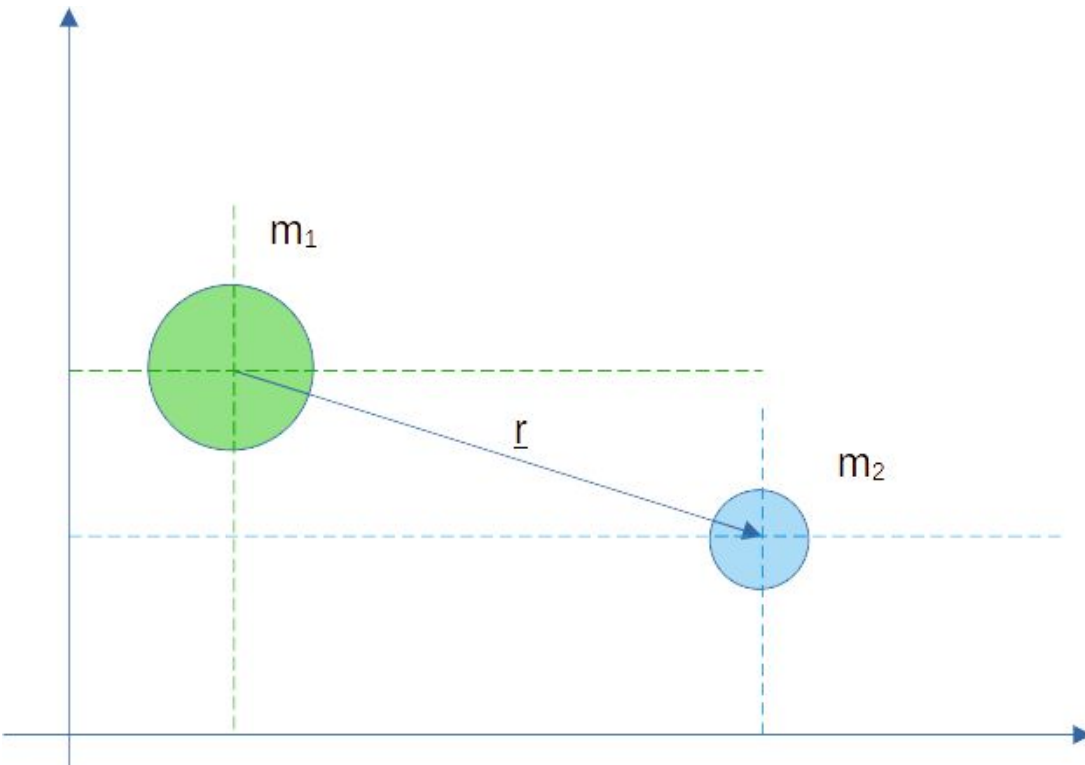
### Three body problem

The three body problem is interesting in the sense that it represents another type of chaotic problem where *slight changes in the initial conditions produce dramatically different outcomes*.

At the same time, there is a level of predictability as each of the bodies in the problem follow a pre-described pattern of motion which is the force of gravity.

$$\hat{\mathbf{f}} = G \frac{m_1 m_2}{r^2} \cdot \hat{\mathbf{r}}$$

In an arbitrary coordinate system of our choosing.



And the fact that the acceleration of a body is proportionate to the force exerted upon it.

$$\hat{\mathbf{f}} = m_i \frac{d^2 \hat{\mathbf{r}}_i}{dt^2}$$

And combining the equations yields the equation of motion of each body.

$$m_1 \frac{d^2 \hat{\mathbf{r}}_1}{dt^2} = G \frac{m_1 m_2}{r^2} \cdot \hat{\mathbf{r}}$$

In fact, we know that 2-bodies behave in an orderly way and sometimes display beautiful behaviour. We could call these orderly patterns, the stable states. They are predictable and the solution deviates towards them just like we saw in the mandelbrot set, where there are areas of convergence.

For the 2-body problem, the stable states are the elliptic (or circular) orbits that the model predicts and that we commonly observe. So it is the introduction of a third body that throws the system into chaos.

It is important to note that our use of the word chaos means *outcomes which are sensitive to initial conditions*. There will be boundary conditions and positions that the chaos cannot produce in the same way that when we looked at the double pendulum, even though we would struggle to predict the position of the lower pendulum at any point in time, we still knew that it exists at a distance  $< L_1 + L_2$  from the attachment point which was guaranteed by the length of the pendulum rods.

So we might expect something similar to be the case for the 3-bodies. So let's get into the code.

We first set up the initial conditions. Our force of gravity,  $g$ , the masses of the 3 bodies along with their starting positions and velocities. The only modules needed were `matplotlib` and `numpy`.

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

```

# gravity
g = 9.8

# masses
m_1 = 10
m_2 = 20
m_3 = 30

# starting coordinates
p1_start = np.array([-10, 10, -11])
v1_start = np.array([-3, 0, 0])

p2_start = np.array([0, 0, 0])
v2_start = np.array([0, 0, 0])

p3_start = np.array([10, 10, 12])
v3_start = np.array([3, 0, 0])

```

Next we make a function that takes the positions and returns back the accelerations. These are taken directly from the laws of motion above.

```
def accelerations(p1, p2, p3):
```

```

def accelerations(p1, p2, p3):
    """
    A function to calculate the derivatives of x, y, and z
    given 3 object and their locations according to Newton's laws
    """
    p_1_dv = -g * m_2 * (p1 - p2)/(np.sqrt((p1[0] - p2[0])**2
        + (p1[1] - p2[1])**2 + (p1[2] - p2[2])**2)**3) - \
        g * m_3 * (p1 - p3)/(np.sqrt((p1[0] - p3[0])**2
        + (p1[1] - p3[1])**2 + (p1[2] - p3[2])**2)**3)

    p_2_dv = -g * m_3 * (p2 - p3)/(np.sqrt((p2[0] - p3[0])**2
        + (p2[1] - p3[1])**2 + (p2[2] - p3[2])**2)**3) - \
        g * m_1 * (p2 - p1)/(np.sqrt((p2[0] - p1[0])**2
        + (p2[1] - p1[1])**2 + (p2[2] - p1[2])**2)**3)

```



```

p_3_dv = -g * m_1 * (p3 - p1)/(np.sqrt((p3[0] - p1[0])**2
      + (p3[1] - p1[1])**2 + (p3[2] - p1[2])**2)**3) - \
g * m_2 * (p3 - p2)/(np.sqrt((p3[0] - p2[0])**2
      + (p3[1] - p2[1])**2 + (p3[2] - p2[2])**2)**3)

return p_1_dv, p_2_dv, p_3_dv

```

Whilst this looks messy, it is actually quite systematic. Each particle's acceleration is calculated based on the mass and position of the other particles in the system.

It is actually at this point where we could extend the same rule to n-bodies. In fact, a problem like this lends itself very well to classes, which arguably would have been a cleaner and more scalable approach. However, for the purpose of the article, we take an explicit approach.

We then set the number of steps and the arrays to contain the data for the steps.

```

# parameters
delta_t = 0.01
steps = 10_000

# initialize trajectory array
p1 = np.array([[0.,0.,0.] for i in range(steps)])
v1 = np.array([[0.,0.,0.] for i in range(steps)])

p2 = np.array([[0.,0.,0.] for i in range(steps)])
v2 = np.array([[0.,0.,0.] for i in range(steps)])

p3 = np.array([[0.,0.,0.] for i in range(steps)])
v3 = np.array([[0.,0.,0.] for i in range(steps)])

```

We can set the starting positions as follows:

```
# starting point and velocity
p1[0], p2[0], p3[0] = p1_start, p2_start, p3_start
v1[0], v2[0], v3[0] = v1_start, v2_start, v3_start
```

And then we are ready to run the simulation and populate the lists which contain the positions and velocities.

```
# evolution of the system
for i in range(steps-1):
    # calculate derivatives
    dv1, dv2, dv3 = accelerations(p1[i], p2[i], p3[i])

    v1[i + 1] = v1[i] + dv1 * delta_t
    v2[i + 1] = v2[i] + dv2 * delta_t
    v3[i + 1] = v3[i] + dv3 * delta_t

    p1[i + 1] = p1[i] + v1[i] * delta_t
    p2[i + 1] = p2[i] + v2[i] * delta_t
    p3[i + 1] = p3[i] + v3[i] * delta_t
```

This is the bulk of the computation process and it takes less than 1 second to process 10,000 steps and likewise 7 seconds for 100,000 steps.

Finally, we are able to see the results of the process with a `matplotlib` plot.

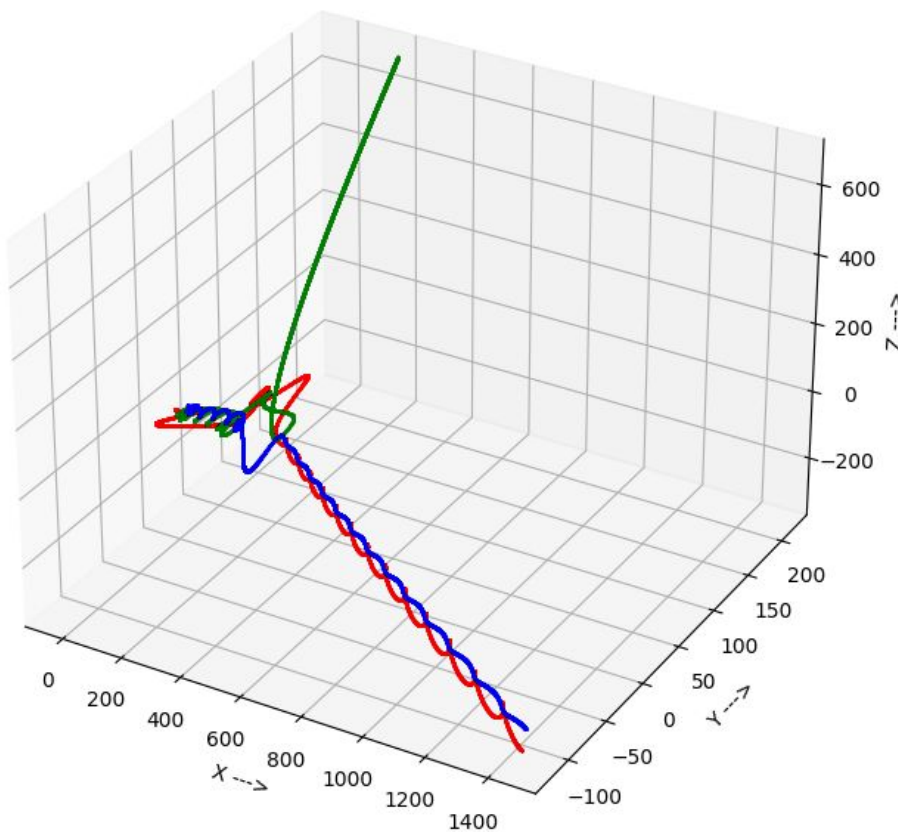
```
# plot the chart
ax = plt.figure(figsize=(8, 8)).add_subplot(projection='3d')

plt.plot([i[0] for i in p1], [j[1] for j in p1], [k[2] for k in p1],
         'r', color='red', lw = 0.05, markersize = 0.1)
plt.plot([i[0] for i in p2], [j[1] for j in p2], [k[2] for k in p2],
         'g', color='green', lw = 0.05, markersize = 0.1)
plt.plot([i[0] for i in p3], [j[1] for j in p3], [k[2] for k in p3],
         'b', color='blue', lw = 0.05, markersize = 0.1)

ax.set_xlabel('X ---->')
ax.set_ylabel('Y ---->')
```

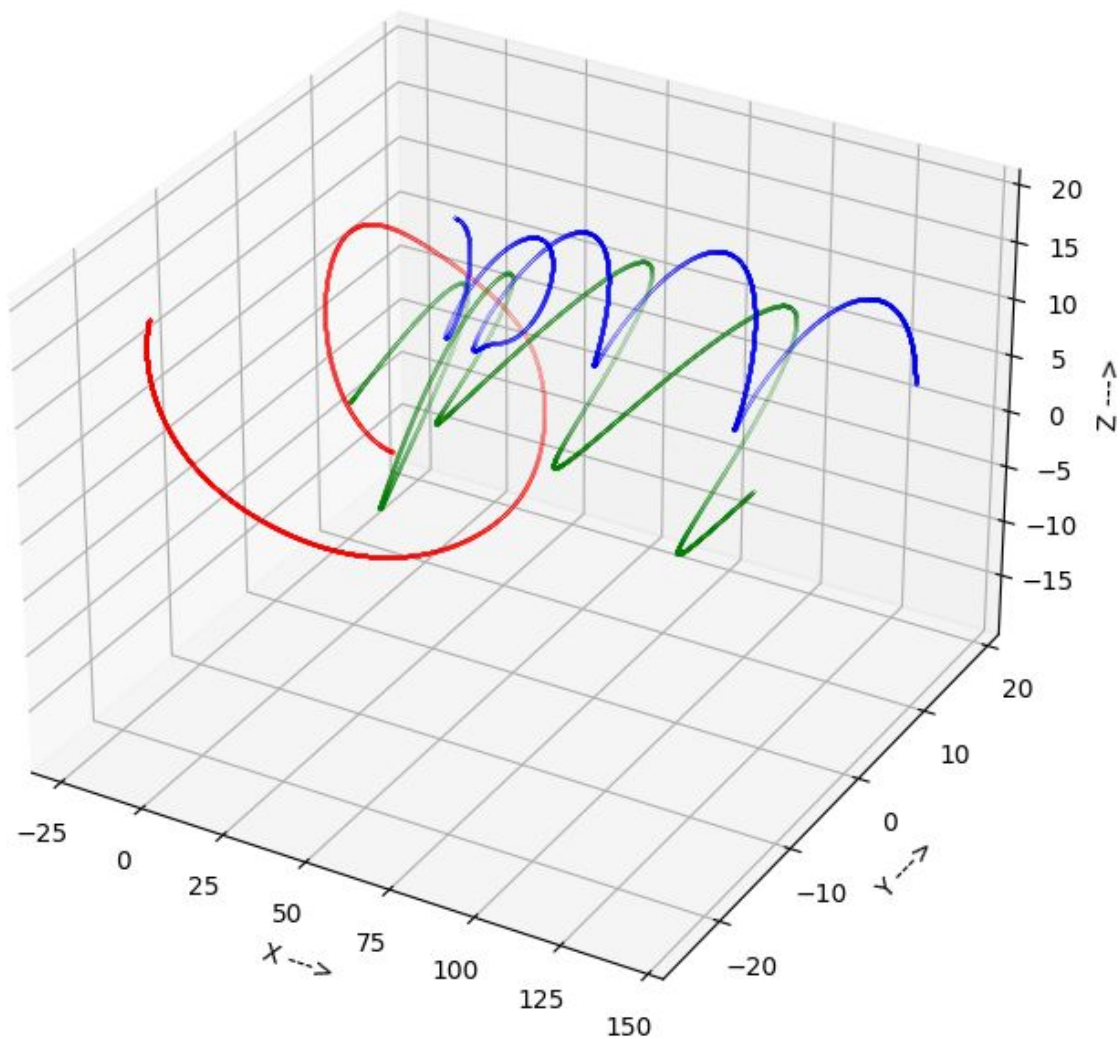
```
ax.set_zlabel('Z --->')  
  
plt.show()
```

Which initially produces this:



Each particle is represented by a colour and we can see that two particles (or planets) co-rotate off with each other whilst the third particle moves in a completely different direction.

The above was for 100,000 loops and some predictability was born out of the chaos, but if we looked at a shorter timescale of the same initial conditions, it would have been difficult to predict that this would have been the case.

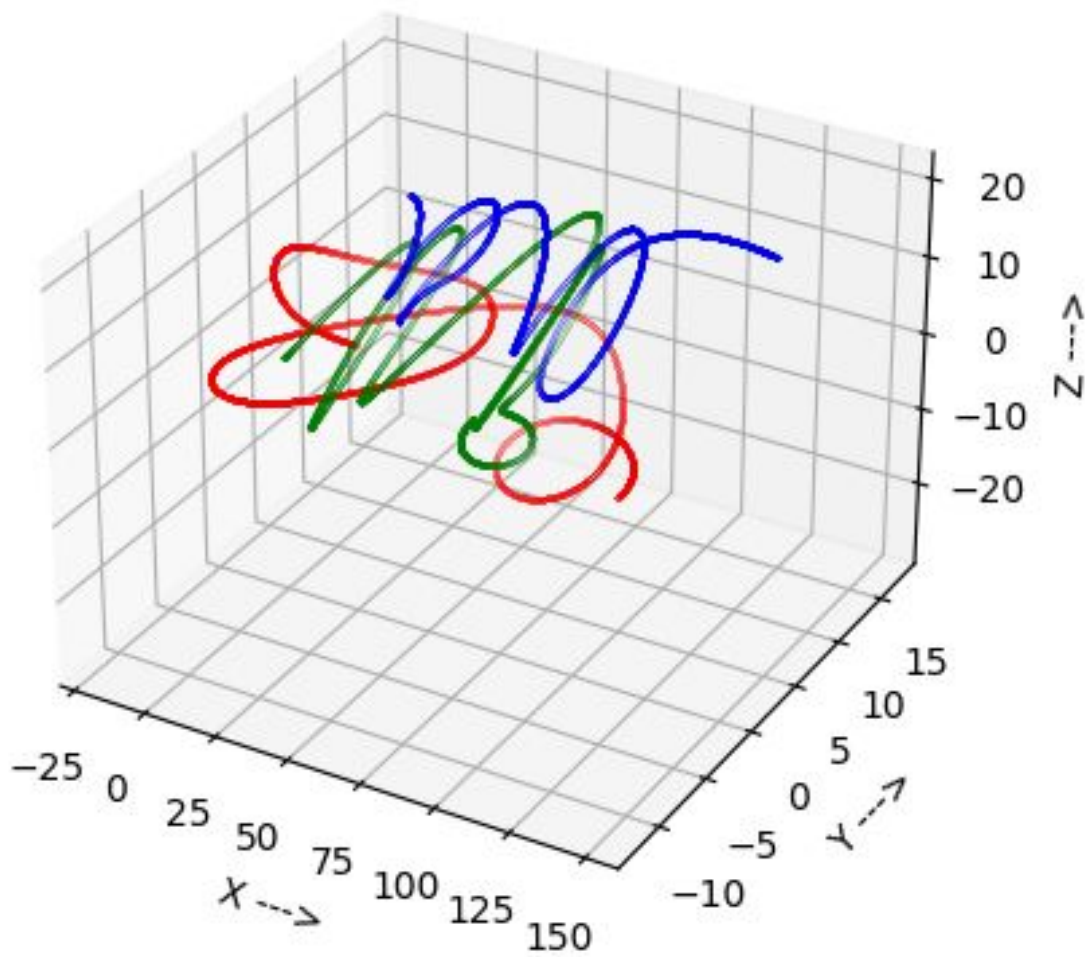


One nice feature of the plot, that we essentially get for free, is that high velocities are indicated by lighter lines because the points are further apart from each other.

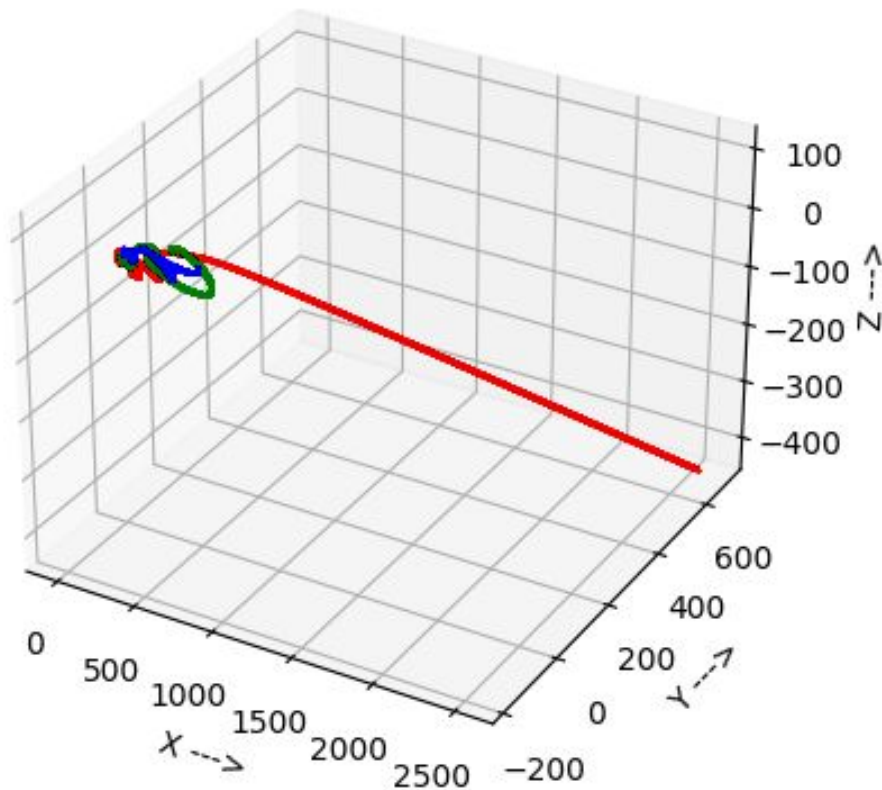
Let's look at what happens if we change the initial conditions by a fraction amount. For example we make this small change to the starting coordinate of particle 1 with no other changes.

```
p1_start = np.array([-10, 10, -11])  
p1_start = np.array([-10, 10, -10])
```

And the resulting short term (10,000 points) chart is completely different:



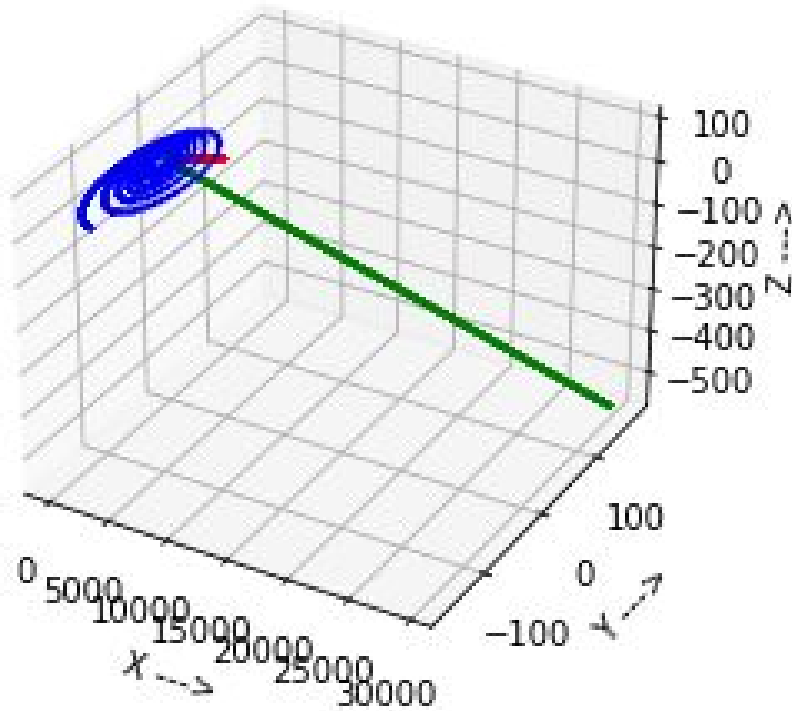
With the longer term chart being radically different. In particular, the red particle is ejected from the system whilst the green and blue co-rotate in



location.

With just the three bodies alone, we can create a huge variety of initial conditions, each one with a completely different outcome. In fact, in writing this, hours were spent in trying to obtain predictable patterns, such as one significantly heavy body with two light ones where we might expect the smaller particles to rotate around the larger one, but in fact we often get ejections from the system.

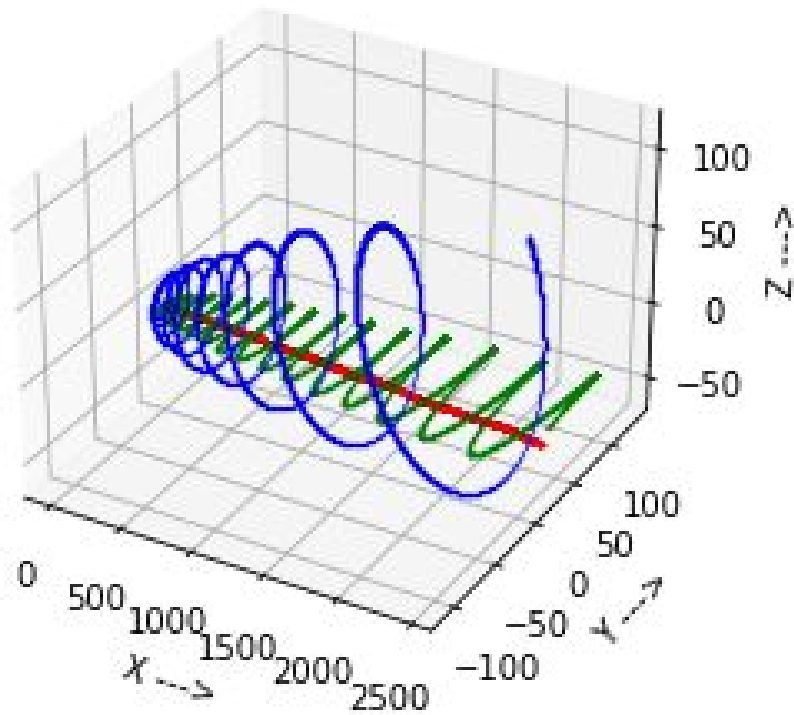
3 body problem  
iterations:25000



Here is a system where the red particle mass is 1000 compared to the blue and the green whose masses are 10. We see that the green particle is ejected from the system.

Just changing the speed of the green particles by 1 metre per second vastly changes the system and it took a lot of care to get a three body pattern like this.

3 body problem  
iterations:25000



Strangely enough, if we were to increase the number of particles in the system, we would actually introduce some kind of measurable statistical order. Probably the subject of a second book !

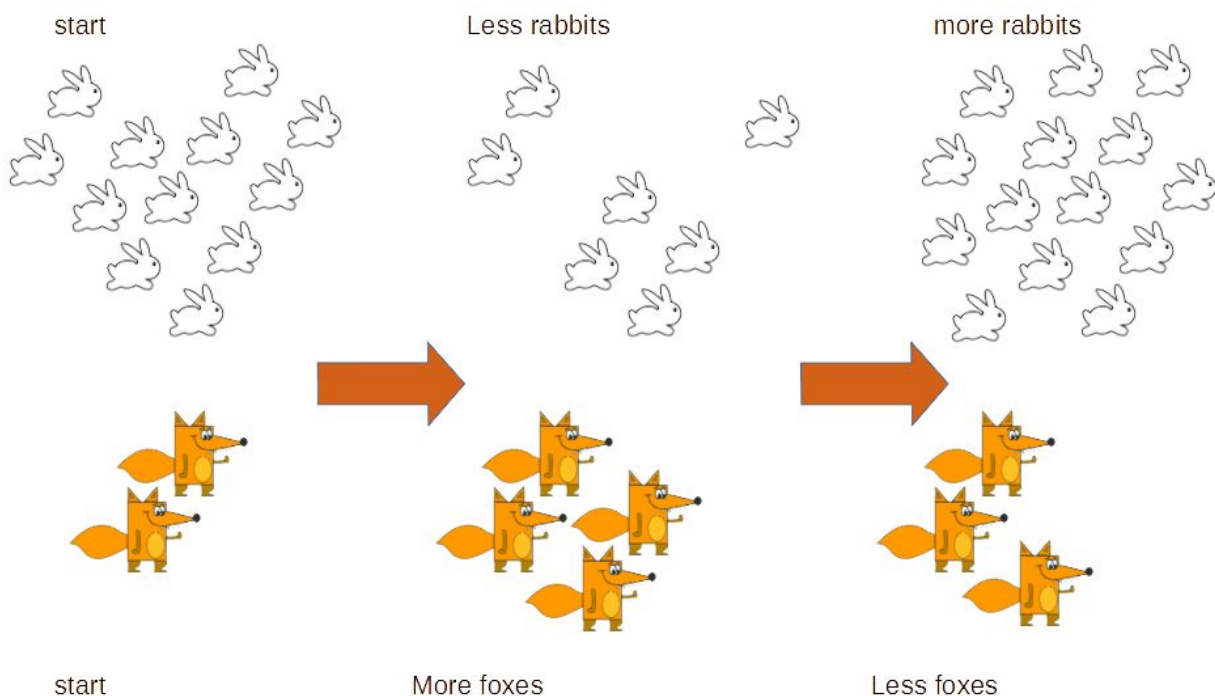


## Predator vs prey cycles

We have spent the last few chapters looking at chaotic systems. We started with the mandelbrot set and then moved into the double pendulum and then onto the three body problem each one producing unpredictable squiggles in both two and three dimensions.

With the predator-prey cycle we expect to see more order. In fact, the behaviour might capture some of the dynamics of diseases and rumours where there is a self dampening effect, although we might expect this to oscillate.

Before we delve into the mathematics, let's consider an overview of a predator prey system. For example foxes and rabbits.



The populations of each component of the system depend on the other population. So more rabbits equates to more food for the foxes which gives rise to an increasing fox population, but more foxes kill the rabbits reducing the rabbit population. This in turn reduces the supply of food for the foxes and subsequently reduces the fox population. Less foxes mean more rabbits ... and so the cycle continues.

The above logic gives rise to the following mathematics:

$$\frac{dx}{dt} = \alpha x - \beta xy$$

$$\frac{dy}{dt} = \delta xy - \gamma y$$

Where x and y are the number of prey and predators respectively.

The parameters  $\alpha$  and  $\gamma$  are interesting as they represent the species dynamics without interaction. So positive  $\alpha$  describes a prey population that naturally increases whilst negative  $\gamma$  describes the opposite for the predators.

This dynamic tends to always be the case as, left alone without predators culling the population, the prey tends to multiply fast. Likewise, left alone essentially with no food, the predator population would decline.

In a similar fashion the connected, xy, terms have the opposite effect on both the populations where we can see that a large xy is positive for the predator population but negative for the prey population.

This is very much the basic mathematics of natural selection in an ongoing ecosystem because if any of those parameters were reversed, then the system would become unstable and at least one of the species would become extinct.

Since we know that:

$$\frac{dy}{dx} = \frac{dy}{dt} / \frac{dx}{dt}$$

We can combine the two equations above to get:

$$\frac{dy}{dx} = \frac{\delta xy - \gamma y}{\alpha x - \beta xy}$$

And rearranging this gives this:

$$\frac{\alpha - \beta y}{y} dy = \frac{\delta x - \gamma}{x} dx$$

Whilst the mathematics was interesting to note, we can actually produce the code from the first two equations. So let's do that.

We start with the standard imports and defining some global parameters.

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

timestep = 0.0001 # determines the accuracy

amp = 0.00 # amplitude of noise term
end_time = 50 # time simulation ends
t = np.arange(0, end_time, timestep) # times vector

# initialise prey (x) and predator (y) vectors
x = []
y = []
```

Next, setting the four coefficients and also the initial conditions for the numbers of predators and prey, making sure that there is sufficient prey, otherwise the system would risk extinction of the prey and therefore collapse.

```
a = 1 # birth rate of prey
b = 0.1 # death rate of prey due to predation
c = 0.5 # natural death rate of predator
d = 0.02 # consumed prey that give rise to predator

# initial conditions at time=0
x.append(100) # prey (x)
y.append(20) # predator (y)
```

We then do the number crunching:

```
# forward euler method of integration
```

```

for index in range(1,len(t)):
    # evaluate the current differentials
    xd = x[index-1] * (a - b*y[index-1])
    yd = -y[index-1]*(c - d*x[index-1])

    # evaluate the next value of x and y
    next_x = x[index-1] + xd * timestep
    next_y = y[index-1] + yd * timestep

    # add the next value of x and y
    x.append(next_x)
    y.append(next_y)

```

At this point we have created and filled the arrays with the population data that we required for the predator and prey populations.

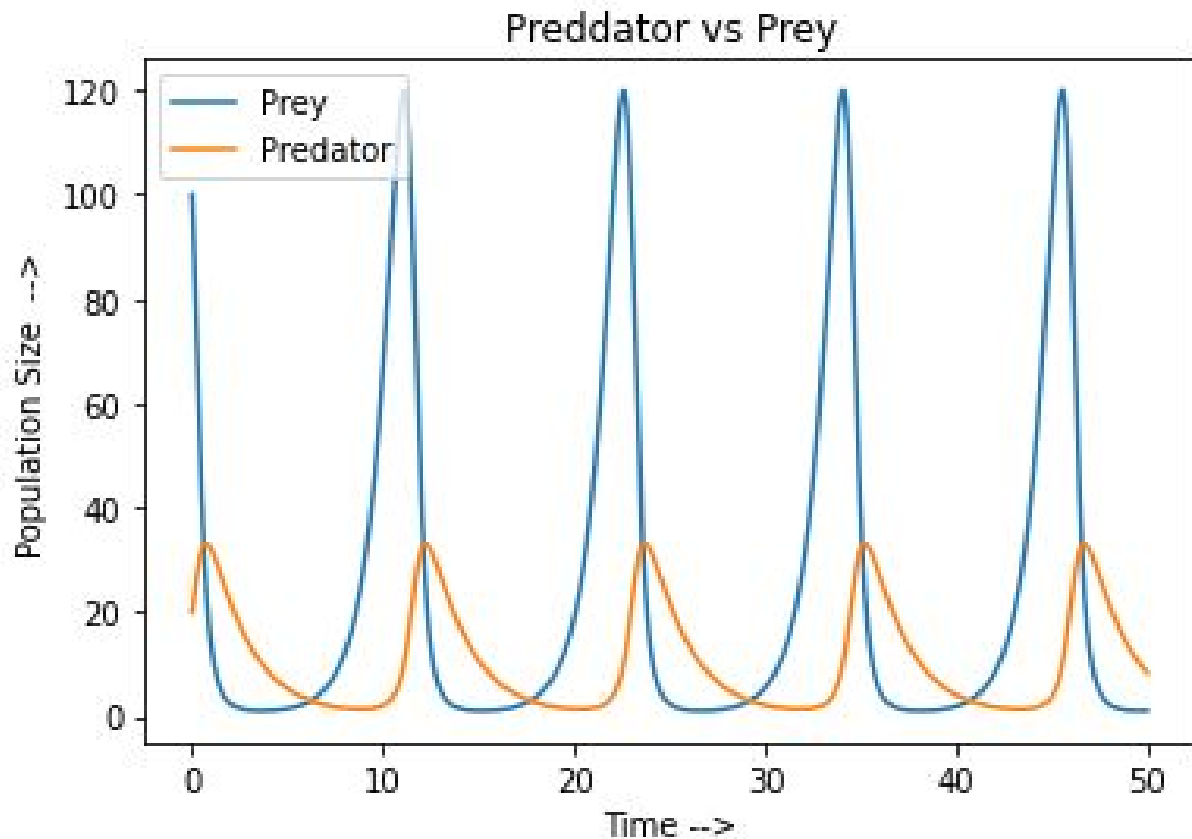
So we can set about plotting the charts.

```

# predator-prey same axis vs time
plt.plot(t, x)
plt.plot(t, y)
plt.xlabel('Time -->')
plt.ylabel('Population Size -->')
plt.legend(('Prey', 'Predator'))
plt.title('Preddator vs Prey')
plt.show()

```

Which gives this:

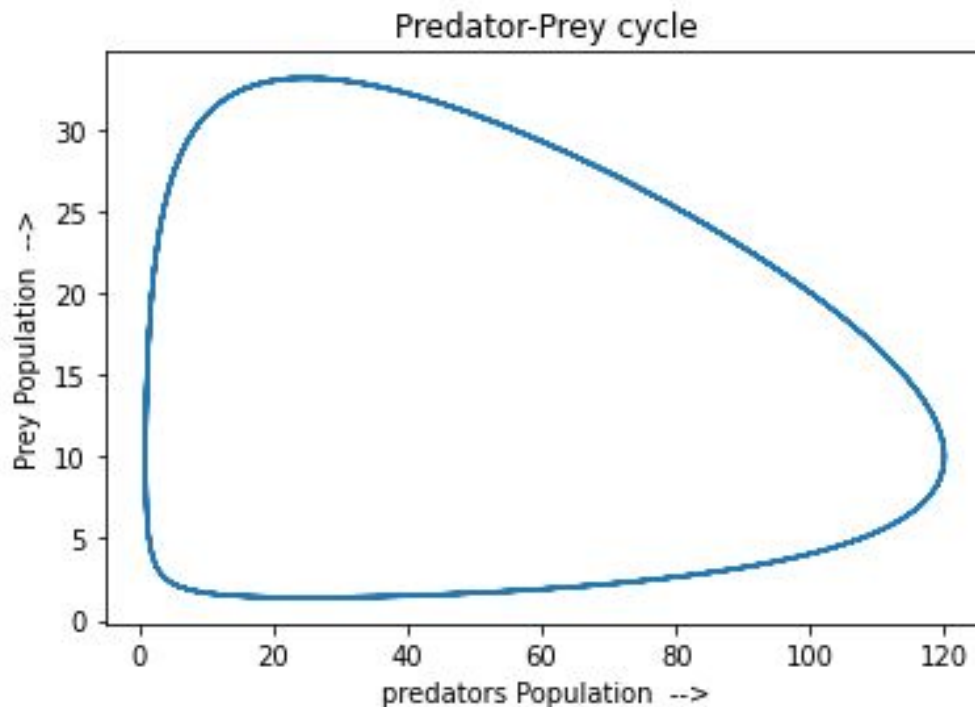


We can see how the cycles for predator and prey are out of phase with each other. Predator lags prey by  $\frac{1}{4}$  of a cycle or  $\frac{\pi}{2}$  radians or  $90^\circ$ .

And then:

```
# predator vs prey cycle
plt.plot(x,y)
plt.xlabel('predators Population -->')
plt.ylabel('Prey Population -->')
plt.title('Predator-Prey cycle')
plt.show()
```

Which gives this:



By plotting predator and prey populations on different axes we can clearly see the cyclical relationship between the two. In particular, when one population is high, the other is low and vice versa.

We can also see how, in both populations the peak to trough variation is quite dramatic and this explains why a difficult period could upset the balance of the ecosystem.

A natural system has stochastic volatility, so we need to introduce this into the model and we can do this with the following code to introduce noise into the system:

```
def StochasticTerm(amp):  
    return (amp * random.uniform(-1,1))
```

We can use this noise to perturb the differential equations by amending the coefficients in the for loop as follows:

```
for index in range(1,len(t)):  
  
    # stochastic parameters  
    a = a + StochasticTerm(amp)
```

```

b = b + StochasticTerm(amp)
c = c + StochasticTerm(amp)
d = d + StochasticTerm(amp)

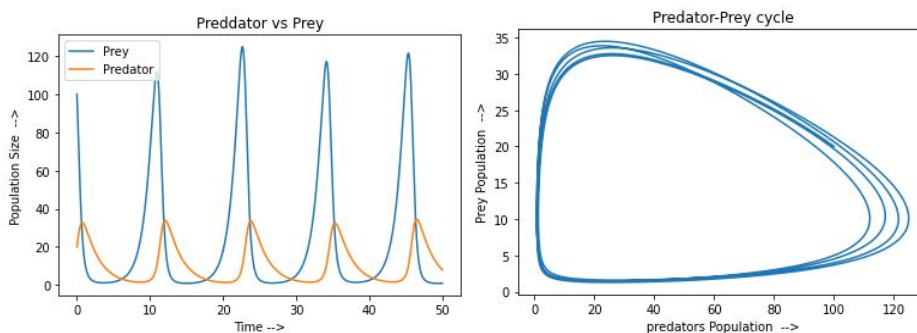
# evaluate the current differentials
xd = x[index-1] * (a - b*y[index-1])
yd = -y[index-1]*(c - d*x[index-1])

# evaluate the next value of x and y
next_x = x[index-1] + xd * timestep
next_y = y[index-1] + yd * timestep

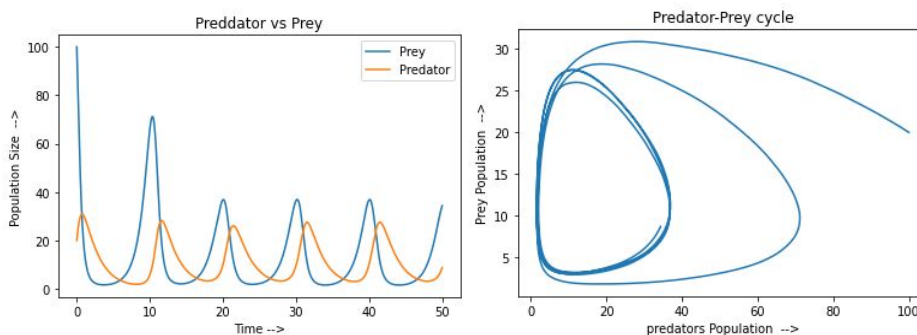
# add the next value of x and y
x.append(next_x)
y.append(next_y)

```

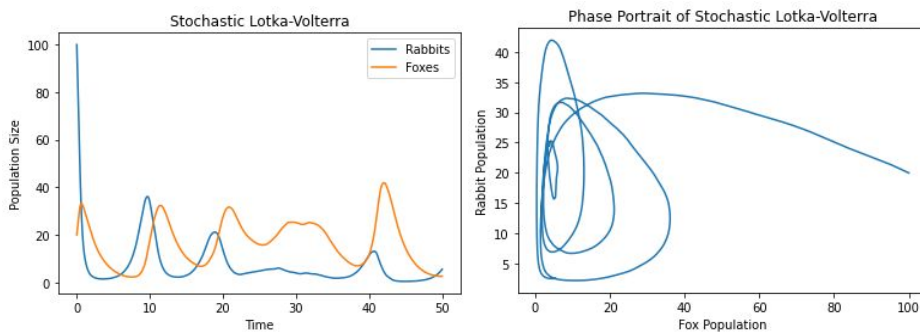
By changing the amp value we are now able to affect the coefficients. A small change in the amplitude ( amp ) from zero to 10 parts in 1 million (0.000001) has this effect.



At 50 parts in 1 million we get this:



And, finally, if the volatility gets too large, the system is unable to repair its steady state and subsequently breaks down.



Just like with the three body problem, there are hundreds of examples and cases to explore and lessons that the models can teach us alongside the mathematics.

And for the purpose of the book, we stop here leaving the rest of the exploring as an exercise for the reader noting just one thing: that ecosystems have many species  $x, y, z \dots$



## Conway Game of life

The game of life is a zero player game, meaning that its evolution is determined by its initial state, requiring no further input.

The user can set the initial state and thereafter the simulation continues indefinitely.

The rules of the game are as follows:

- Each of which is in one of two possible states, **live** or **dead** (**on** or **off** or **1** or **0**).
- Every cell interacts with its eight neighbours

	NW	N	NE	
	W	C	E	
	SW	S	SE	

At each step in time, the following is calculated for the next iteration.

1. Any live cell with *fewer than two live neighbours* dies, as if by underpopulation.
2. Any live cell with *two or three live neighbours* lives on to the next generation.
3. Any live cell with *more than three live neighbours* dies, as if by overpopulation.
4. Any dead cell with *exactly three live neighbours* becomes a live cell, as if by reproduction.

So the evolution of a cell is contingent on the neighbouring cells in all cases.

We can set up these rules in python and create an animation in matplotlib:

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

```

# globals
OFF, ON = 0, 1
vals = [ON, OFF]

def main():
    """ main function """
    N = 100 # set grid size
    updateInterval = 100 # set animation update interval

    # declare grid
    grid = np.array([])
    grid = randomGrid(N)

    for i in range(5):
        addGlider(np.random.randint(2,N-2),
                  np.random.randint(2,N-2), grid)

    # set up animation
    fig, ax = plt.subplots()
    img = ax.imshow(grid, interpolation='nearest')
    ani = animation.FuncAnimation(fig, update,
                                  fargs=(img, grid, N, ), frames = 10,
                                  interval=updateInterval,save_count=50)

    plt.show()

def randomGrid(N):
    """returns a grid of NxN random values"""
    pyes=0.04
    pno=1-pyes
    return np.random.choice(vals, N*N, p=[pyes, pno]).reshape(N, N)

def addGlider(i, j, grid):
    """adds a glider with top left cell at (i, j)"""
    glider = [[0,0,1],[1,0,1],[0,1,1]]

```

```

grid[i:i+3, j:j+3] = glider

def update(frameNum, img, grid, N):
    # copy grid since we require 8 neighbors
    newGrid = grid.copy()
    for i in range(N):
        for j in range(N):

            # compute 8-neighbor sum (x and y wrap around)
            total = int(grid[i, (j-1)%N] + grid[i, (j+1)%N] +
                        grid[(i-1)%N, j] + grid[(i+1)%N, j] +
                        grid[(i-1)%N, (j-1)%N] + grid[(i-1)%N, (j+1)%N] +
                        grid[(i+1)%N, (j-1)%N] + grid[(i+1)%N, (j+1)%N])

            # apply Conway's rules
            if grid[i, j] == ON:
                if (total < 2) or (total > 3):
                    newGrid[i, j] = OFF
            else:
                if total == 3:
                    newGrid[i, j] = ON

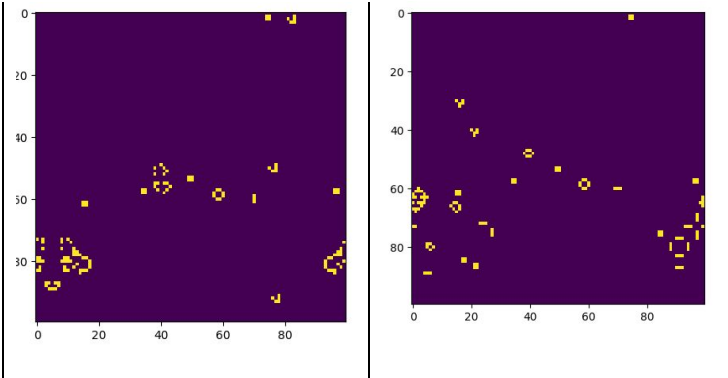
    # update data
    img.set_data(newGrid)
    grid[:] = newGrid[:]
    return img

# call main
if __name__ == '__main__':
    main()

```

There are a huge number of variants and twists that one could add to this most simple of games, but before we consider those, let's take a look at the output of the code.

--	--


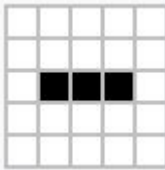
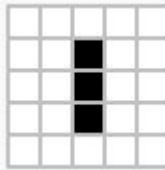
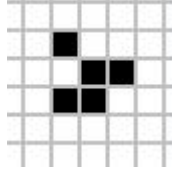
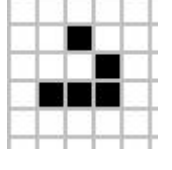


The resulting observation is continually evolving patterns with some common features that have been investigated over the years.

In particular, the following three stand out:

- Still lifes - never change (unless collided with)
- Oscillators - don't move, but flip between states.
- Spaceships - Oscillators that travel in a constant direction.

Conceptualising the above shapes is comparatively easy when the size is small. Here are some base examples:

Still life	oscillator	spaceship
<p>block</p> 	<p>blinker</p>  	<p>Glider</p>  

We were able to make a simple glider with this block of code:

```
def addGlider(i, j, grid):
    """adds a glider with top left cell at (i, j)"""
    glider = [[0,0,1],[1,0,1],[0,1,1]]
    grid[i:i+3, j:j+3] = glider
```

There are many more intricate shapes of this nature such as guns which produce gliders and a lot of work has been done into what appeared to be a rather trivial game initially.

In another separate piece of code, displaying different techniques (pygame in place of matplotlib animation) we are able to see an example of a gun.

```
import pygame
import numpy as np

col_about_to_die = (100, 200, 225)
col_alive = (255, 255, 215)
col_background = (10, 10, 40)
col_grid = (30, 30, 60)

def update(surface, cur, sz):
    nxt = np.zeros((cur.shape[0], cur.shape[1]))

    for r, c in np.ndindex(cur.shape):
        num_alive = np.sum(cur[r-1:r+2, c-1:c+2]) - cur[r, c]

        if cur[r, c] == 1 and num_alive < 2 or num_alive > 3:
            col = col_about_to_die
        elif (cur[r, c] == 1 and 2 <= num_alive <= 3) or \
            (cur[r, c] == 0 and num_alive == 3):
            nxt[r, c] = 1
            col = col_alive

        col = col if cur[r, c] == 1 else col_background
        pygame.draw.rect(surface, col, (c*sz, r*sz, sz-1, sz-1))

    return nxt

def init(dimx, dimy):
```

```
cells = np.zeros((dimy, dimx))

pattern = np.array([

    [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
    [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
    [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
    [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
    [1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
    [1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
    [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
    [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
    [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
    [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]]

pos = (3,3)

cells[pos[0]:pos[0]+pattern.shape[0],
      pos[1]:pos[1]+pattern.shape[1]] = pattern

return cells
```

```
def main(dimx, dimy, cellsize):

    pygame.init()

    surface = pygame.display.set_mode((dimx*cellsize, dimy*cellsize))

    pygame.display.set_caption("Game of Life")

    cells = init(dimx, dimy)

    while True:

        for event in pygame.event.get():

            if event.type == pygame.QUIT:

                pygame.quit()

                return

            surface.fill(col_grid)

            cells = update(surface, cells, cellsize)

            pygame.display.update()
```

```
if __name__ == "__main__":

    main(120, 90, 8)
```

Whilst this is just a game, it allows the user to try out a number of techniques, indeed, we could have used tkinter (or even turtle) for the GUI with similar implementation.

We note the different colouration for cells which are about to die, which are those that will be removed in the next iteration.

We also note that the starting grid is explicitly defined where the pattern of the glider cannon (the gun) is saved in the variable named pattern.

The output is an animation, but the pattern for the gun is static:



A follow up of the game of life is left to the reader...

## The Lorenz Attractor

An exercise for the reader.

## Digital Image Processing

A huge amount of work has been done over the past 25 year with respect to image processing and the growth of the subject and technologies arising from such have been tremendous.

Image processing can often be processor intensive and as such has greatly benefitted from the increase in processing power over the same period.

Python has proved to be a popular language for image processing and comes with a number of libraries that intrinsically contain the techniques that this section will be looking at. In particular, it is Python's ability to harness and leverage the much faster C++ algo's that has provided this popularity.

One of the more popular modules that Python, called `opencv` , was originally written in C++. Whilst there are a number of other modules, we will focus on this one.

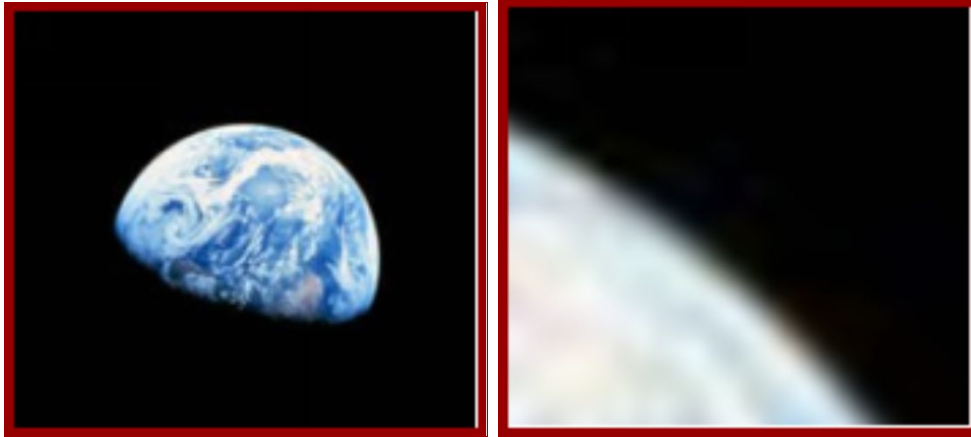
In this chapter we will attempt to understand a number of the processes leading up to and finally including face detection or other interesting digital equivalents. We will start at the beginning here:

- Read images, videos, webcams
- Basic functions
- Resizing and cropping

### Introduction to images

Even though we might not be able to see this with the naked eye, computer images that we see on our screens, monitors or other devices are made up from many small points called pixels. When we zoom into the images this becomes noticeable as the image becomes more and more pixelated.

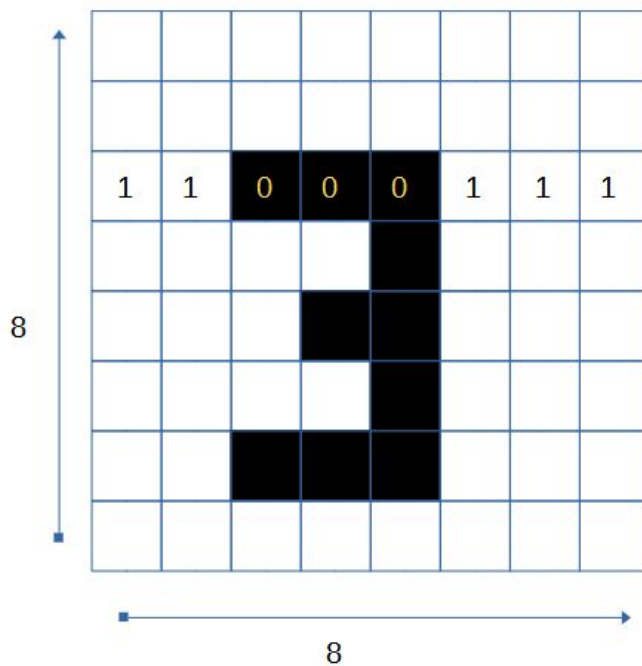




So we can consider an image as a series of squares.

1	2	3	4	5	
6	7	8	9	10	
11	12	13	14	15	
16	17	18	19	20	
21	22	23	24	25	

And we can keep on zooming in to those squares until we reach the resolution of the pixels themselves. So imagine that we wanted to draw the number “3”, it would look something like this:



All the black boxes are given the value 0 and all of the white boxes are given the value 1. The image itself is 8x8 squares (or pixels) width and height. So 64 pixels in total.

Screen sizes are understood to be in terms of ( width x height ) where each parameter represents the number of pixels in each dimension. Typical screen sizes are:

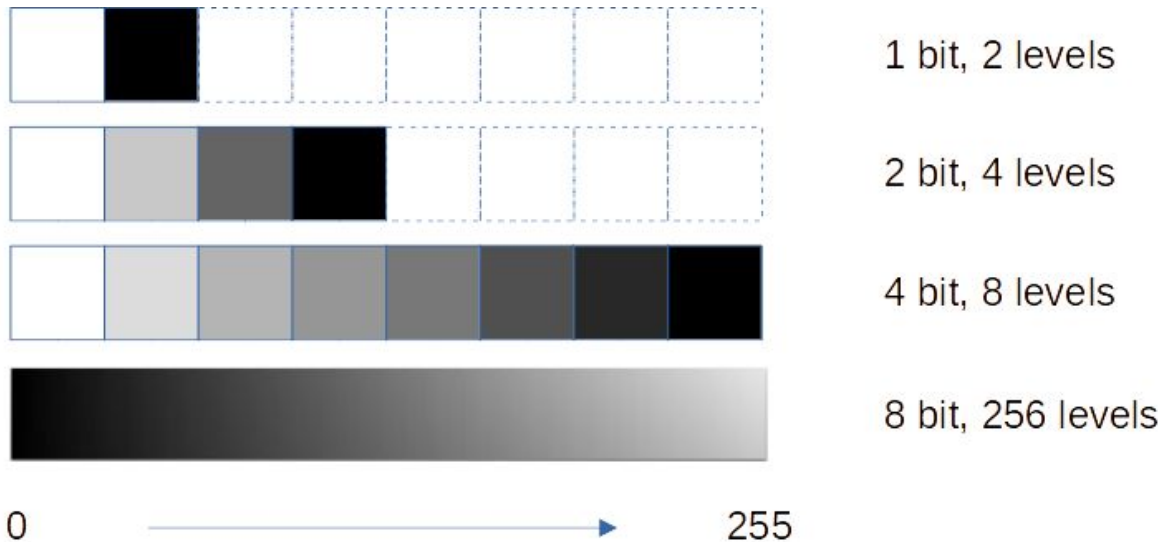
- VGA: 640 x 480
- HD: 1280 x 720
- 4k: 3840 x 2160

So an HD display has 1280 pixels going across and 720 pixels in the vertical direction. So there are 921,600 pixels in this display. Since the human eye cannot resolve each dot of the nearly 1 million dots in a VGA image, this is sufficiently sharp for highly detailed images to be displayed.

The image above is a binary image. It can either contain a black square or a white square, but no other colour.

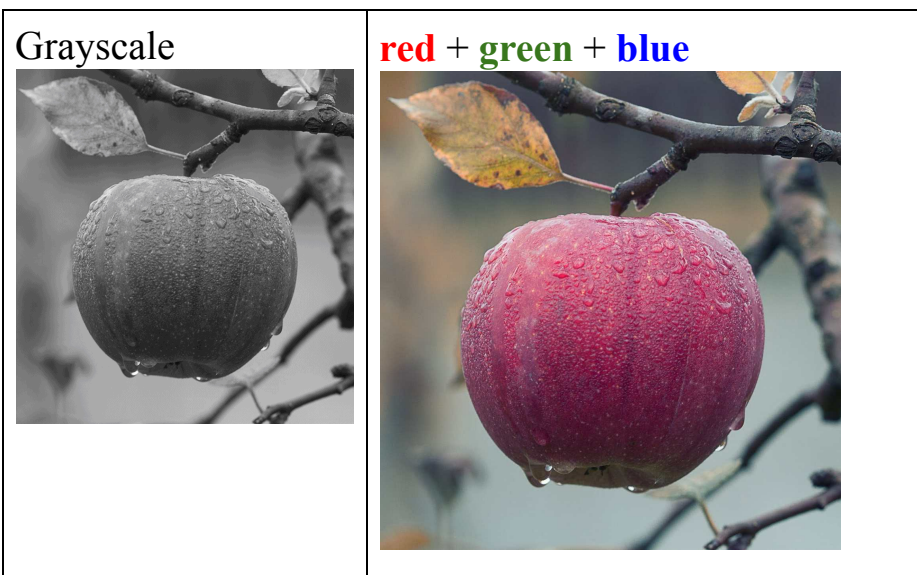
- 0 = Black
- 1 = white

But we could improve this by allowing for different numbers to create a grayscale in place of the basic black and white images.



The different shades of grey make the image clearer. The scales go up in powers of 2 (because they are binary) and the current number of levels that gives a good image is  $2^8$  which is 256 levels.

A colour image is made up in exactly the same way as grayscale images with the only difference being that there are 3 scales: **red** + **green** + **blue**. And the sum of these gives the image that we see. We know this as RGB.



So a coloured HD image is now 1280 x 730 x 3 (a pixel for each colour). Each one of these 921,600 pixels is made up from a sum of 3 colours each with 256 levels of intensity.

## Reading images and videos

Now that we have a basic understanding of how images are made we can begin to move a bit deeper into the process.

One of the nice things about Python is the ease of use that it provides. Importing an image is done with the `opencv-python` module which is a third party module like the `pandas` and `numpy` modules and is quickly installed with the package manager ( `pip install opencv-python` ).

With this module it is easy to read an image:

```
import cv2

img = cv2.imread('assets/apple_colour.png')
cv2.imshow('Output', img)
cv2.waitKey(0)
```

The entire image is read into the memory and then displayed on the screen.

We can inspect the contents of the image like this:

```
print(img)
print('columns:', len(img))
print('rows:', len(img[0]))
print('colours:', len(img[0][0]))
```

And we see this:

```
[[[ 64 104 129]
  [ 64 104 129]
  [ 64 104 129]
  ...
  [ 88  73  70]
  [ 88  73  70]
  [ 88  73  70]
  ...
  ...
  [194 194 178]
  [193 193 175]
```

```
[192 192 174]]]
```

```
columns: 1204
```

```
rows: 1164
```

```
colours: 3
```

This is a list of lists (an array) with each sub list containing the RGB values. So, our example image (of the coloured apple) contained a list of 1204 elements (the height) x 1164 elements (the width) x 3 elements (the red, green and blue values). So describing this image was over 4 million values. Whilst we don't get to appreciate the brilliance of the image in a book, the blown up image on a HD screen is spectacular.

Importantly, we now have the data in just a few lines of code and this is valuable.

Swiftly moving on from images, we can capture a video (by creating a video capture object) like this:

```
import cv2

cap = cv2.VideoCapture('assets/sample_video.mkv')

while True:
    done, img = cap.read()
    if done == False:
        break
    cv2.imshow('Video', img)
    if cv2.waitKey(1) & 0xFF == ord('q'):
        break
```

The first thing that we note is that a video is made up of a series of images. The `cap.read()` function returns a tuple with the first value being if the image was read successfully and the second value the image file itself. We have seen list unpacking before (in the data science chapter) and this is now a convenient feature.

The complicated `waitkey(1)` line is essentially waiting for the escape key or the letter “q” to be pressed and could be replaced by this:

```
key = cv2.waitKey(1)
```

```
if key == 27 or key == 113:  
    break
```

We like to keep things simple.

So, we were able to read and display an image file, then we were able to read and display a video file which was a series of images. Our last technique is to capture a video from the device webcam. So this is a *real-time* video as opposed to a pre-stored video.

Reading data from the webcam exactly the same as the video with the exception of 1 line of code changing into 3 lines:

```
cap = cv2.VideoCapture(0)  
cap.set(3,640) # width  
cap.set(4,480) # height
```

The VideoCapture() function reads zero as the *default web camera* of the device. And the two set() functions are the image size (width and height) that we want to return.

## Basic functions

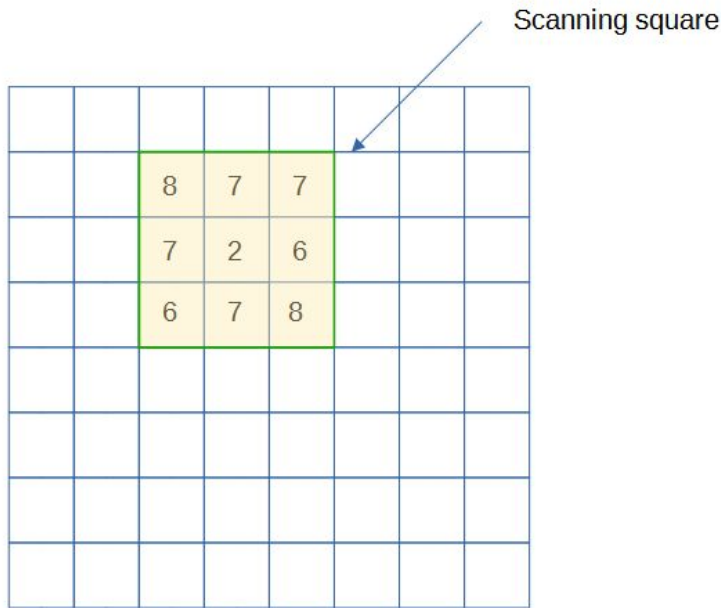
In the above section, we were able to:

- Read an image file and display the picture.
- Read a video file and display the video.
- Read the device camera and display the video.

The cv2 module in python offers a lot of image processing functions out of the box. We will try to understand them first and then implement a few.

The general rule for image processing is to send a scanning square across the image. With this we can do a lot of features such as blurring and edge enhancements, sharpening etc. we could even do colour filters and so forth.

If we wanted to blur or smooth the image then we could take an average of all of the pixels in the square and assign the average to the central pixel.



In the example above the average of all of the 9 pixels in the scanning square are 6 rounded to the nearest integer. We assign this value to the central pixel, so its value changes from 2 to 6. And as we move this square across the image in the i and j direction in a nested for loop or list comprehension, averages are generated for all of the pixels in the image. We have smoothed the image. There are a few things to note in the process.

- The size of the scanning square (smoothing and processing time)
- The average at the edges (information loss).
- Different types of averaging (different effects and complexities).

Likewise, it is possible to do edge enhancements by comparing adjacent columns of pixels in the list.

There are many applications ranging from fashion, media and fun to military and industrial equipment.

### **Blur image:**

The process for blurring images is effectively to take an average of the pixels in the array around the target pixel. The cv2 module offers a Gaussian blur which would be different to a simple average type process.

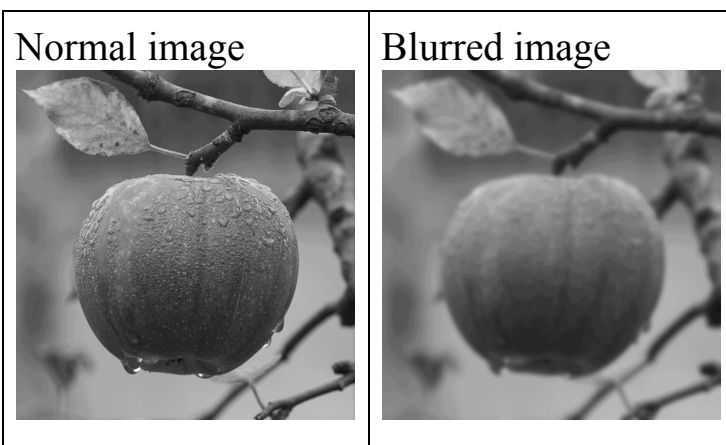
Here is the code:

```
import cv2
img = cv2.imread('assets/apple_colour.png')
img_gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
img_blur = cv2.GaussianBlur(img_gray, (49,49), 0)

cv2.imshow('Gray image', img_gray)
cv2.imshow('Blur image', img_blur)

cv2.waitKey(0)
```

And this is the result:



In the code we did two processes:

1. Convert the image to grey (we could have retained colour).
2. Blurred the image.

An interesting side note is that the opencv module refers to colours as BGR (Blue, Green, Red) in place of the well known convention of RGB. This is the elephant in the room for opencv whose core modules were created in the early days and the BGR convention has stuck, so we have to get used to this.

### Edge detector:

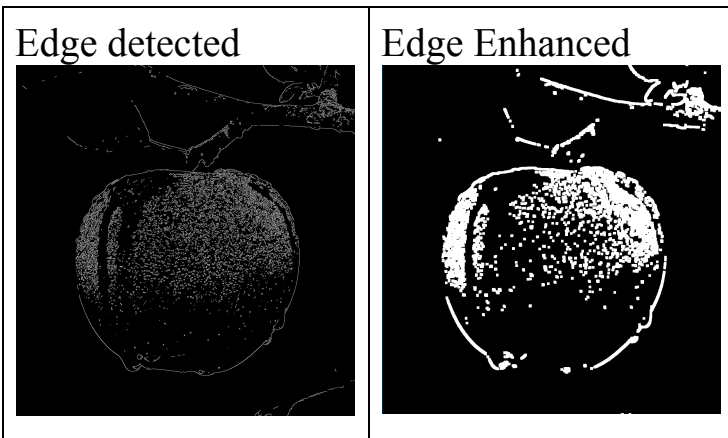
For edge detection we need just two additional lines of code:

```
img_edge = cv2.Canny(img_gray,150,200)
cv2.imshow('Edge image', img_edge)
```

And if we want to enhance the edges more, then we can dilate the image. With this extra code:



```
import numpy as np
kernel = np.ones((5,5), np.uint8)
img_dilation = cv2.dilate(img_edge, kernel, iterations=2)
cv2.imshow('dilated image', img_dilation)
```



We can see how we are not able to see the sides of an object and even enhance these. This could be useful for avoiding bumping into a chair or staying in the centre of a road.

As a final note, it is possible to do the reverse of a dilation, which is known as erosion like this:

```
img_eroded = cv2.erode(img_dilation, kernel, iterations=2)
cv2.imshow('eroded image', img_eroded)
cv2.waitKey(0)
```

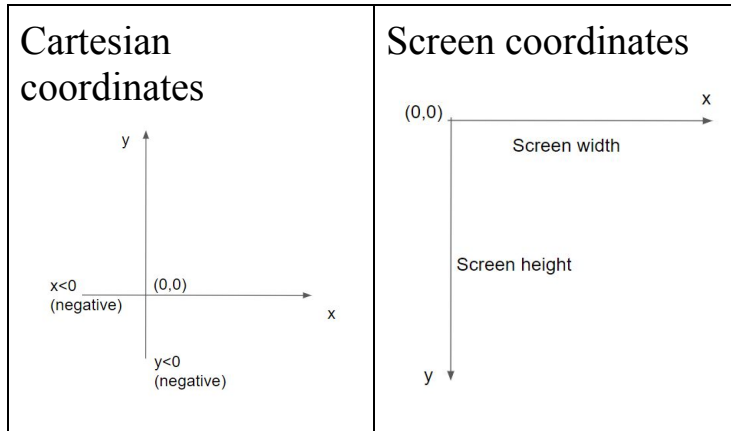
And erosion of iterations=2 is not equal to a dilation of the same order in an analogous way to 90% not being the opposite to 110% (think about that for a second). To undo an operation, it is cleaner to return to the previous operation.

## Resizing and cropping

Images in opencv and other similar applications tend to define the coordinate system from the top left hand corner of the screen. That means

that the top left is the point of origin ( $x=0$ ,  $y=0$ ) .

This is different from the standard cartesian coordinate system that we are used to which has the origin in the bottom left hand corner and can also have negative values.



Negative numbers are “off the screen”. Also numbers greater than width and height exist outside the boundary of the visible screen, so would not be drawn or shown.

Most drawing packages tend to follow the same coordinate system and we have already experienced the same in the GUI sections (for turtle, tkinter and pygame).

Now that we understand how to represent an image, we can dive into the code with our example image:

```
import cv2

img = cv2.imread('assets/apple_colour.png')

h, w, c = img.shape
print(img.shape)
print(f'height: {h}, width: {w}, colours: {c}')
```

We are able to get the shape using the shape property, which returns a tuple containing the height, width and number of colours (BGR) in our case.

Our code returned this:

```
(1204, 1164, 3)
```

```
height: 1204, width: 1164, colours: 3
```

So now we have the dimensions of the picture.

We can now use the `resize()` method to change the dimensions of the image, such as shrinking or stretching it by adding the following code:

```
img_resize = cv2.resize(img, (100,100))  
img_stretch = cv2.resize(img, (500,200))  
  
cv2.imshow('resized image', img_resize)  
cv2.imshow('resized image', img_stretch)  
  
cv2.waitKey(0)
```

Noting that the `resize` takes a tuple (width, height) which was a different order to the shape property of (height, width, colour).

So with opencv, we need to be careful of the order of the dimensions and colours.

Here is the resulting image:



After resizing (stretching) and image, we may also want to crop and image. This is useful when the user wants just one particular area of interest from a much larger image, for example a face of one individual from a crowd of people or a licence plate of a car amid the traffic etc.

For cropping, we do not require a method in the module because we can take advantage of our knowledge that an image is a matrix (which is a list of lists), so we can just extract the area of interest by slicing the list.

Python natively offers a useful feature for lists called slicing which works as follows:

```
x = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']  
x[3:6]  
  
# returns this: ['d', 'e', 'f']
```

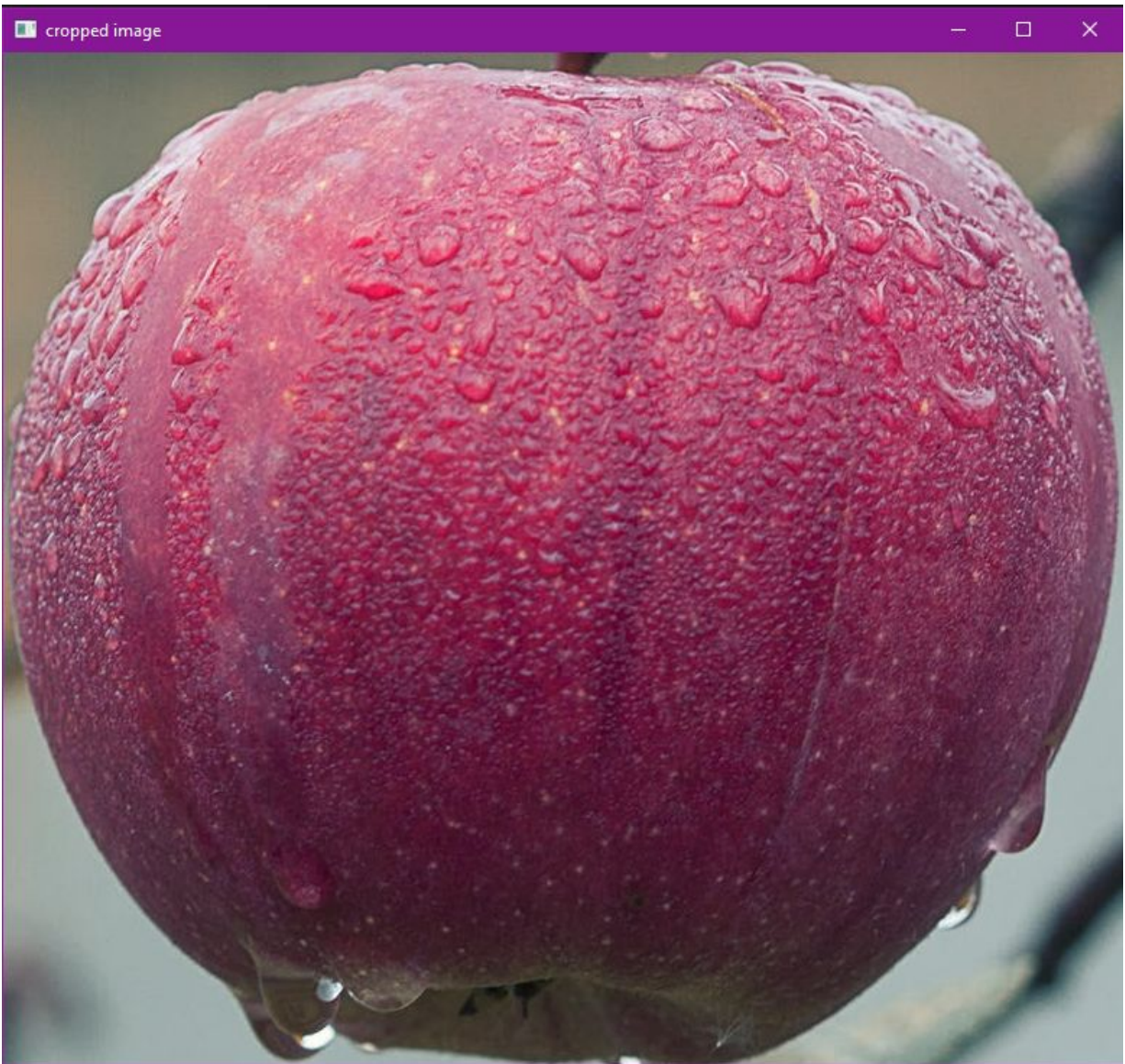
Given a list, we can extract a portion (*called a slice*) of a list from a start index (remembering that indices start from zero) to an end index (but not including the end index itself).

So in the example above `x[3:6]` returns the letters in index positions 3, 4 and 5. Which are 'd', 'e', 'f'.

So using this convenient and pythonic method would just need this:

```
img_cropped = img[350:1050, 200:975]  
cv2.imshow('cropped image', img_cropped)
```

Here we have cropped (by slicing) elements 350 to 1050 in the vertical direction (y-axis) and elements 200 to 975 in the horizontal direction (x-axis). Which results in selecting only the apple from the original image.



The resulting image is also smaller in size by virtue of a new smaller height and width.



## Shapes and text

This section covers the basics of adding various lines and shapes to an image. We recall that the image is just a matrix of numbers.

We can create a square image of 512 x 512 and assign each element of the image with a zero value remembering that zero represents black (no colour).

Inside this square black image we create a long purple (purple is blue mixed with red) rectangle.

```
import cv2
import numpy as np

img = np.zeros((512, 512, 3)) # black square

# purple rectangle, height=200, width=20
img[100:300, 200:220] = (255,0,255)

cv2.imshow('image', img)
cv2.waitKey(0)
```

We use the same slicing technique as we used from the cropping section to create the rectangle.

To this image, we add a green line starting at the origin (0,0) and going to the bottom right corner (512,512).

```
# green diagonal line
cv2.line(img, (0,0), (512, 512), color=(0,255,0), thickness=2)
```

We then add an yellow rectangle

```
# yellow rectangle
cv2.rectangle(img, (100,50), (250,250),
              color=(0,255,255), thickness=10)
```

The rectangle is defined in a similar way to the line. All the user was required to do was input the coordinates of the top left and bottom right ordinates of the rectangle shape.

Next we add a white circle remembering that white is a maximum of all the colours mixed (255,255,255). We also need to know that the ordinates that

we enter define the centre of the circle and its radius. When there are lots of arguments, it is sometimes clearer to use the keywords that the arguments are assigned to.

```
# white circle
cv2.circle(img, center=(350,350), radius=100,
           color=(255,255,255), thickness=50)
```

And finally adding some text in red saying “hello world”:

```
# add some text in red
cv2.putText(img, text='Hello World', org=(200,100),
            fontFace=cv2.FONT_HERSHEY_PLAIN,
            fontScale=3, color=(0,0,255), thickness=2)
```

And the entire combination of all of the above components look like this:



At this point we have sampled some of the basics of what we can do to append an image and it is left as an exercise for the reader to be creative with this.

## Warp perspective

This is a useful technique for changing the perspective of an image so that we can get a birds eye view of the *flattened* image. Imagine for example that we take a picture of a document at an angle, but we want to see its true rectangular shape.



Required card

Or in our example case above, we have some playing cards scattered on the table and we want to get the king of spades. We would need to know the coordinates of the corners of the card to do this which we can get by hovering over the image.

```
import cv2
import numpy as np

img = cv2.imread('assets/cards.jpg')

width, height = 200, 300
points = np.float32([[82,157],[212,134],[115,356],[264,323]])
```



```
new_pts = np.float32([[0,0], [width,0], [0,height], [width,height]])  
matrix = cv2.getPerspectiveTransform(points, new_pts)  
img_warp = cv2.warpPerspective(img, matrix, (width, height))  
  
cv2.imshow('output image', img_warp)  
cv2.waitKey(0)
```

Using the old points and the new points one can produce the transformation matrix. This is done in the cv2 module with the function `getPerspectiveTransform()` and using this matrix one can then proceed to perform the warp retaining the perpendicular flattened image.



## Colour detection

When it comes to colour detection, the logical step would be to select colours by their RGB (or BGR) tuple values. For example, if we wanted to pick blues, then these would be all tuples where the blue value was relatively high compared to the other values.

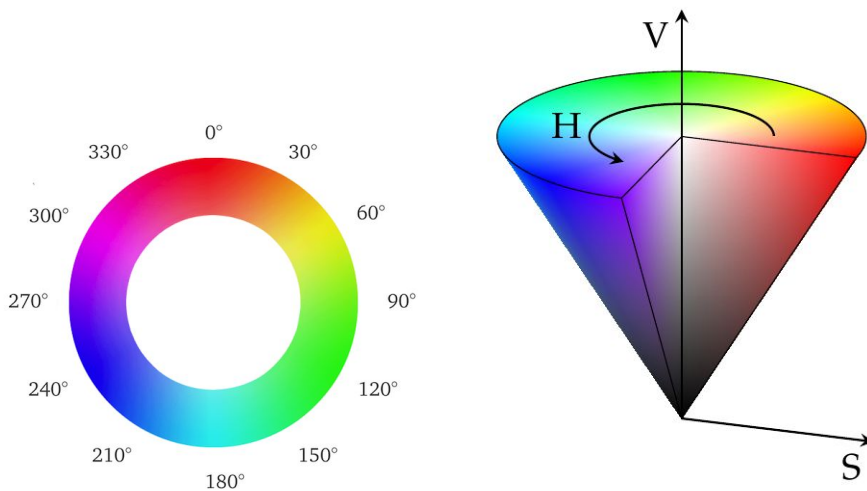
So (200, 50, 50) in the opencv module would be predominantly blue. We could therefore, by this logic, select all colours that were blue in nature. The

blue ordinate  $> 200$  would be sufficient.

However, whilst this logic could be used, the preferred method is to convert the image to HSV or Hue Saturation Value which is used to separate image luminance from colour information. This makes it easier when we are working on or need luminance of the image/frame. HSV is also used in situations where colour description plays an integral role.

In the HSV representation of colour, hue determines the colour you want, saturation determines how intense the colour is and value determines the lightness of the image.

So we up with a transformation that looks like this:



There are other similar representations of the same, but with this type of model, the colours are split into their categories in a more convenient way. For example one can obtain all the reds which include the light and dark reds, but seeking a hue that is close to 0 degrees, say  $< 30^\circ$  and  $> 330^\circ$  on the polar scale.

In OpenCV, Hue has values from 0 to 180, Saturation and Value from 0 to 255. Thus, OpenCV uses HSV ranges between (0-180, 0-255, 0-255).

So with our original apple image we could transform the BRG image to an HSV image. With the HSV image we then create a mask image and finally and extract the reds from this using the mask as follows:

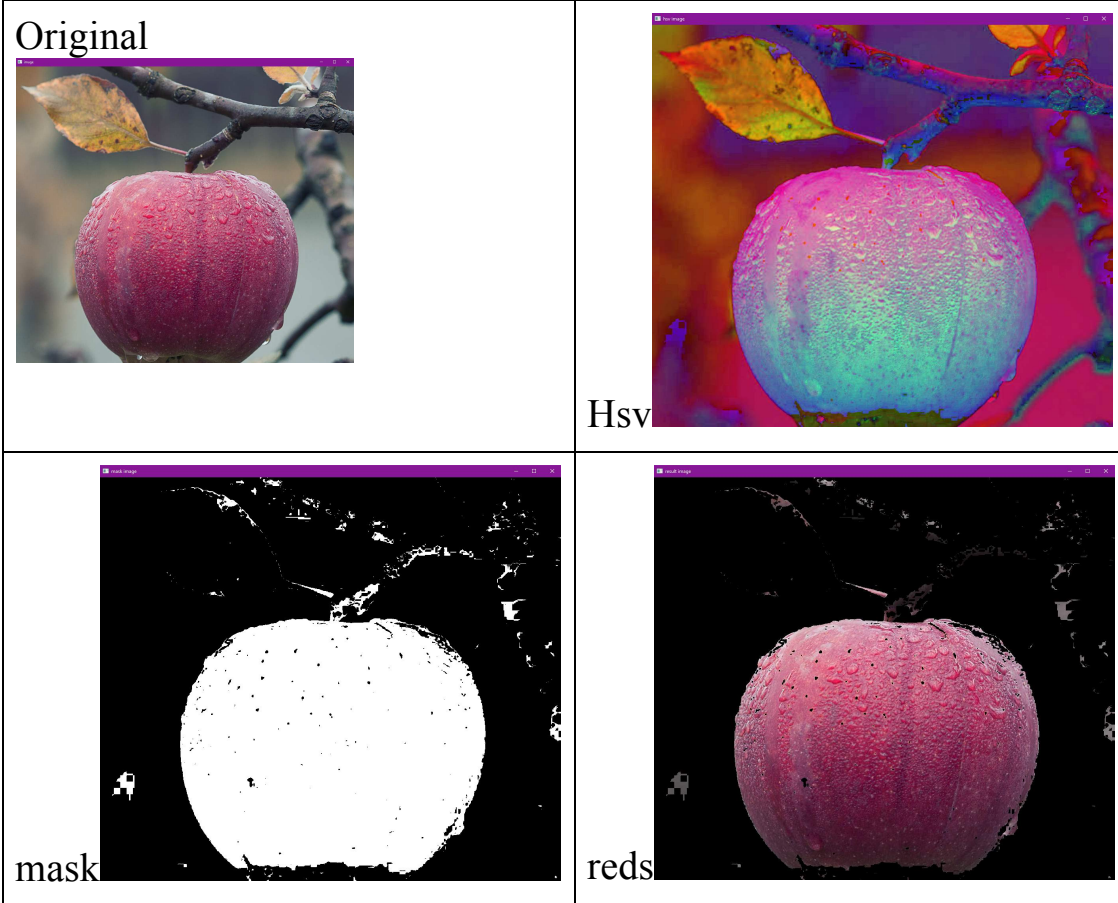
```
import cv2
```

```
img = cv2.imread(base_path + 'assets/apple_colour.png')
img_hsv = cv2.cvtColor(img, cv2.COLOR_BGR2HSV)
hue_low, hue_high = 160, 180 # hugh for reds
img_mask = cv2.inRange(img_hsv, (hue_low,0,0), (hue_high, 255,255))
img_result = cv2.bitwise_and(img, img, mask=img_mask)

cv2.imshow('image', img)
cv2.imshow('hsv image', img_hsv)
cv2.imshow('mask image', img_mask)
cv2.imshow('result image', img_result)
cv2.waitKey(0)
```

And here are the resulting images.

Original → hsv → mask → reds



We see that the HSV technique was convenient in extracting the reds from the red apple picture.

## Shape detection

In order to detect shapes, it is useful to have a method for detecting edges. Fortunately we have touched upon such methods in the previous sections and in particular in the basic functions we were able to produce a crude edge enhancement for the apple image.

The opencv module provided some useful functions for the above:

- GaussianBlur() → smoothing
- Canny()<sup>[25]</sup> → edge detection
- Dilate() → edge enhancement

The above enabled a smoothing of the original image with the GaussianBlur , and then edge detection with the Canny() function and finally, enhancement of the edges with the Dilate() function.

```
import numpy as np
import matplotlib.pyplot as plt
import cv2

img = cv2.imread('assets/shapes.jpg')

# blur the image
img_blur = cv2.medianBlur(img, 5)

# convert to grayscale
img_gray = cv2.cvtColor(img_blur, cv2.COLOR_BGR2GRAY)

# perform edge detection
img_edge = cv2.Canny(img_gray, 30, 100)

# detect lines in the image using hough lines technique
img_line = cv2.HoughLinesP(img_edge, 1, np.pi/180, 60, np.array([]), 50, 5)

# # finds the circles in the grayscale image using the Hough transform
img_crc1 = cv2.HoughCircles(image=img_gray,
                             method=cv2.HOUGH_GRADIENT, dp=0.9, minDist=80,
                             param1=110, param2=39, maxRadius=70)

# iterate over the output lines and draw them
for line in img_line:
    for x1, y1, x2, y2 in line:
        cv2.line(img, (x1, y1), (x2, y2), (20, 100, 20), 10)

# iterate over circles
for co, i in enumerate(img_crc1[0, :], start=1):
    # draw the outer circle in green
    cv2.circle(img, (int(i[0]),int(i[1])),int(i[2]),(0,255,0),5)
```

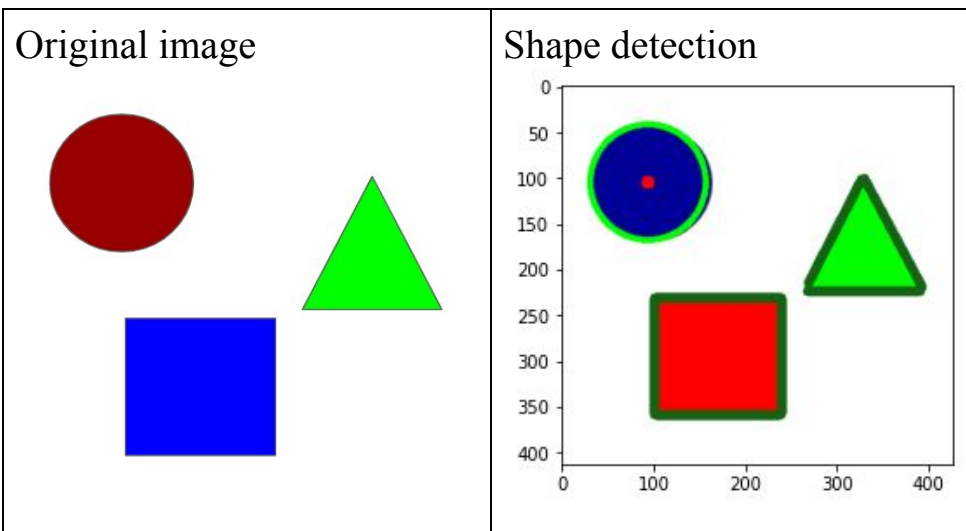
```
# draw the center of the circle in red
cv2.circle(img,(int(i[0]),int(i[1])),2,(255,0,0),10)

# show the image
plt.imshow(img)
plt.show()
```

It was necessary to detect (and draw) the lines and circles using different techniques:

- HoughLinesP
- HoughCircles

But otherwise, it was possible to perform shape detection of different shape types in less than 50 lines of code. So let's create an image with a circle, square and triangle and then invoke the above routine.

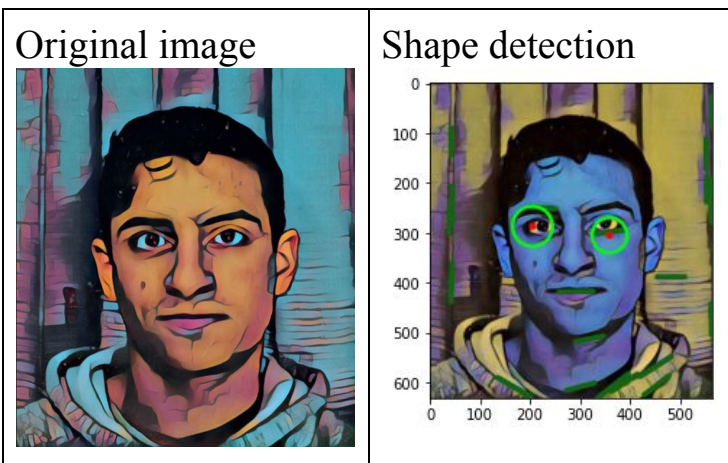


We were able to identify the various shapes successfully with the python opencv module. Note that it was also possible to show the locations and dimensions of the shapes as well as the type of shape and even its colour.

Facial recognition

Facial recognition relies on all that we have learnt above in the previous chapters. We will need [1] shape detection to recognise where faces might be and [2] an AI solution with some form of trained data that contains faces (and not faces).

Interestingly, the code run on a face image does have some crude performance even though it is light on code. This is the result.



In the above, there is a detection of the eyes via the circle detection method and although not completely accurate (by visual inspection), there is even an attempt at the centre of the eye.

So with the above, we have hope that the methods deployed are leading us down the correct path towards a solution.

Fortunately, the opencv module provides a file which is pre-trained for faces, so the user does not need to run a training file beforehand as would otherwise have been necessary in absence of such data.

In fact opencv has provided a number of default cascades that can detect different things such as number plates, eyes, or even full bodies. Using the cascades leaves the code clean:

```
import cv2

img = cv2.imread('assets/author_02.jpg')
```

```

# Load the cascade
face_cascade = cv2.CascadeClassifier(
    "haarcascade_frontalface_default.xml")

# Convert into grayscale
img_gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

# Detect faces
faces = face_cascade.detectMultiScale(img_gray, 1.1, 4)

# Draw rectangle around the faces
for (x, y, w, h) in faces:
    cv2.rectangle(img, (x, y), (x+w, y+h), (0, 0, 255), 10)

# show the image
cv2.imshow('result', img)
cv2.waitKey(0)

```

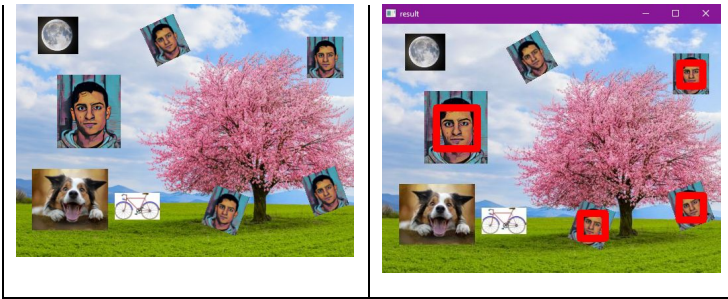
And the result is clear cut for a single facial image.



In fact we can make a complicated image by adding random items and a background and perform the same test.

Before	After





It is observed that the algo is also good for different sizes and also rotations of the faces (of upto 20 degrees) whilst eliminating all the background items.

## Virtual painting

So we have come a huge way on the journey of image processing. The start was to identify an array of white and black squares and then we did various basic functions to change the image features and finally ended up with facial recognition.

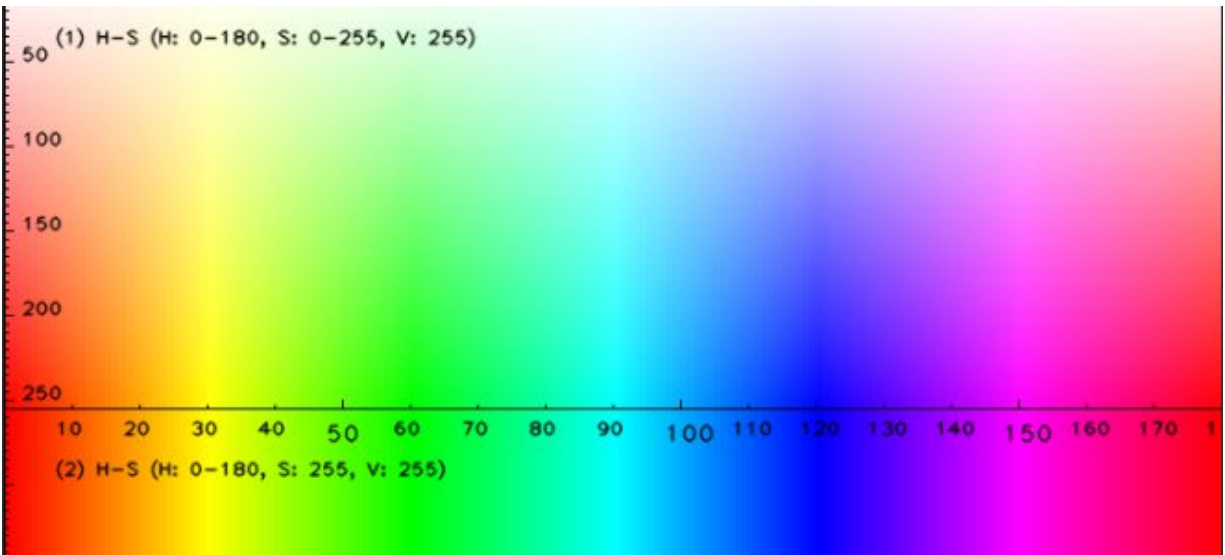
There are a multitude of applications that one could conceive, but let's try a virtual painting application where we can hold up any pen colour to the camera and draw an image on the screen.

For this task, we will need to find colours of our pens using the webcam and subsequently place points on the screen at the tip of the pen so that we can draw a virtual image.

We will use some of our existing knowledge and code to assist with the task at hand.

The first thing that we need to know is the HSV range for selecting colours. The alternative method is to use colour pickers to detect the colour of our pens.

The general range is this:



So we can establish greens to be greens to have a hue-range of 40 to 80 and blues to have a hue-range of 100 to 140. This will be enough to pick our pens. If we wanted red, then we would need two ranges with huge = 0 to 10 for the orangey reds (lighter reds) and 170 to 180 for purple reds (darker reds).

So knowing how to deploy a video capture method and now knowing how to select colours for our pens we can generate this code:

```
import cv2
import numpy as np

print('starting video capture...')
cap = cv2.VideoCapture(0)

# screen dimensions
w, h = cap.get(3), cap.get(4)
print('video shape', w, h)

# calibrated from a color picker.
lower_blue = np.array([100,150,0])
upper_blue = np.array([140,255,255])
lower_green = np.array([40,150,0])
upper_green = np.array([80,255,255])
```

```

my_colors = {'blue': {'lower': lower_blue, 'upper': upper_blue},
             'green': {'lower': lower_green, 'upper': upper_green}}

def find_color(img, my_colors:dict):
    img_hsv = cv2.cvtColor(img, cv2.COLOR_BGR2HSV)
    for color in my_colors.keys():
        lower = np.array(my_colors[color]['lower'])
        upper = np.array(my_colors[color]['upper'])
        mask = cv2.inRange(img_hsv, lower, upper)
        cv2.imshow('image:'+color, mask)
    return

while True:

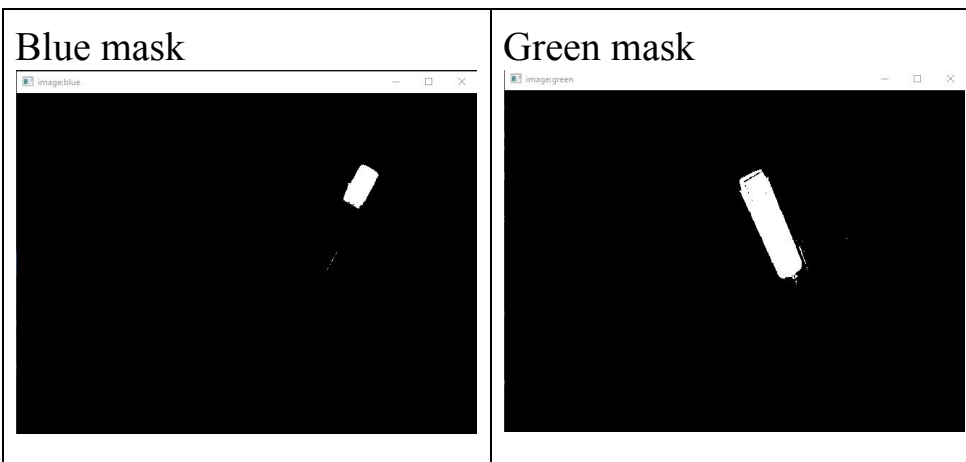
    # break at the end
    done, img = cap.read()
    if done == False:
        break

    find_color(img, my_colors)
    cv2.imshow('Video', img)

    # break if 'esc' or 'q' is pressed
    key = cv2.waitKey(1)
    if key==27 or key==113:
        break

```

Notice that the code allows for the main image and each colour that we elect to add to the mask. So three windows will open.



For each of the masks that we have detected, we need to find where the object is in the image. For this we need to find the contours and approximate a bounding box around it so that we can find the location of the object.

For this, we can create a contours function like this:

```
def get_contours(img):  
    contours, hierachy = cv2.findContours(img,  
                                           cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_NONE)  
    for cnt in contours:
```

```

area = cv2.contourArea(cnt)
print(area)
if area>500:
    cv2.drawContours(img_result, cnt, -1, (255,0,0), 3)
    peri = cv2.arcLength(cnt, True)
    approx = cv2.approxPolyDP(cnt, 0.02*peri, True)
    x,y,w,h = cv2.boundingRect(approx)

```

This basically enables us to show contours around the pen colours.



The result appears to be satisfactory as we can see the pens highlighted with the red borders. So the next task is to identify the tops of the areas as this is where we intend to draw on the screen.

We can do this by adding a return value to the contour function that we created earlier with this line of code.

```

return x+w//2, y + h//10

```

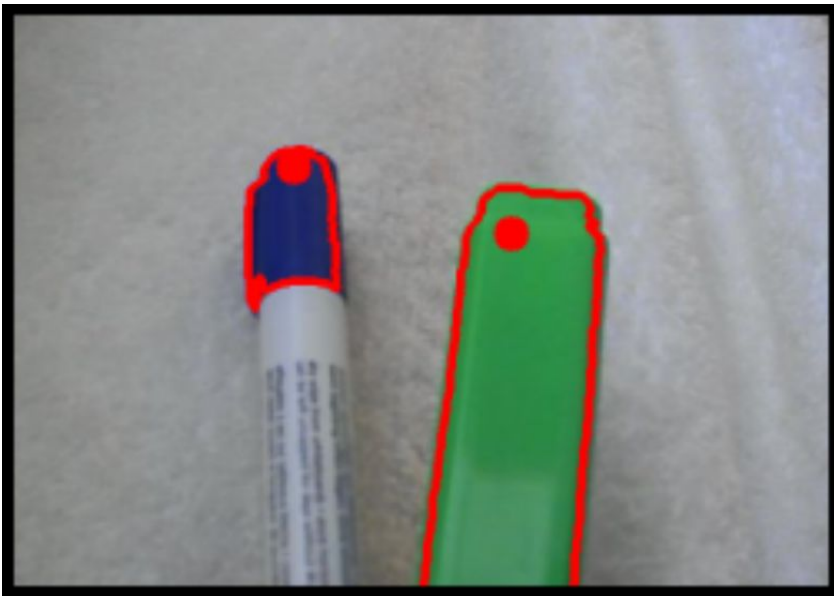
The  $x + w/2$  is the top left + half the width, which is the centre point across. Likewise for the vertical we use  $y$  and one tenth of the height so that

we are near the tip. (remembering that the height starts from the top going downwards in most of the drawing modules).

And we can parse these returned values back into the `find_color()` function and draw circles at these points by adding this code:

```
x, y = get_contours(mask)
cv2.circle(img_result, (x,y), 10, (0,0,255), cv2.FILLED)
```

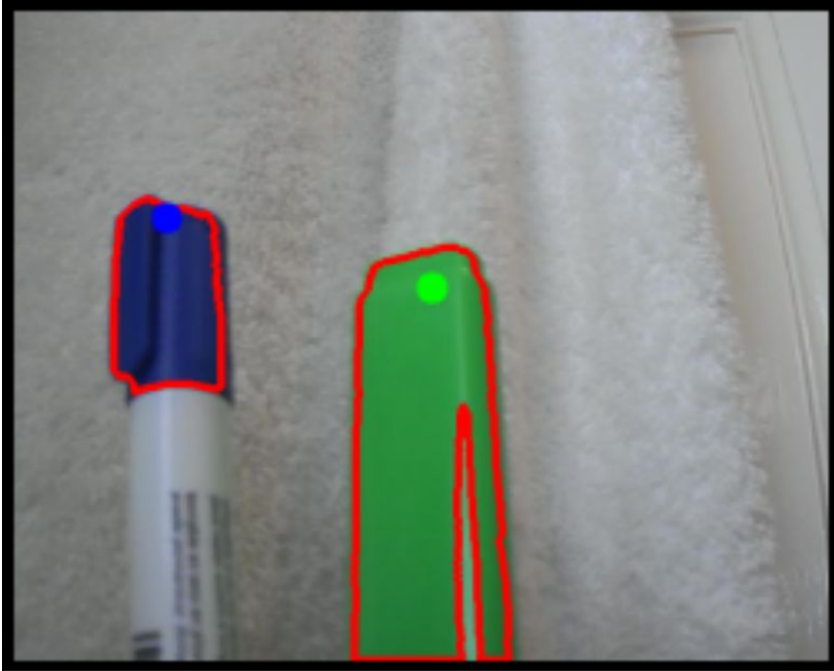
Which again yields a satisfactory result.



The next stage is to change the red circles to the colours of the pens. And the easiest way to do this is to extend the `my_colors` dictionary to include a colour for each pen.

```
my_colors = {'blue':
    {'lower':lower_blue, 'upper':upper_blue, 'value':[255,0,0]},
    'Green':
    {'lower':lower_green, 'upper':upper_green, 'value':[0,255,0]}}
```

This means that we have conveniently parsed the values back into the `find_color()` function as part of the dictionary. Dictionaries are an excellent tool for this type of use as they are extendable and fast.



The final task is to capture these dots into a list so that we can store and draw them onto the screen. The list will need to contain the (x, y) position on the screen and also its colour. So each item in the list will be a tuple like this: (x, y, colour).

And we will need to create a draw function to draw the items that are stored in the list onto the canvas which is done with this function:

```
def draw_on_canvas(my_points):  
    for p in my_points:  
        cv2.circle(img_result, (p[0], p[1]), 10, p[2], cv2.FILLED)
```

For completeness we show the entire code which is less than 80 lines in length.

```
import cv2  
import numpy as np  
  
print('starting video capture...')  
cap = cv2.VideoCapture(0)  
  
# screen dimensions  
w, h = cap.get(3), cap.get(4)  
print('video shape', w, h)
```

```

# calibrated from a color picker.
lower_blue = np.array([100,150,0])
upper_blue = np.array([140,255,255])
lower_green = np.array([40,150,0])
upper_green = np.array([80,255,255])

my_colors = {
    'blue': {'lower': lower_blue, 'upper': upper_blue, 'value': [255,0,0]},
    'green': {'lower': lower_green, 'upper': upper_green, 'value': [0,255,0]} }

my_points = []

def find_color(img, my_colors=dict):
    img_hsv = cv2.cvtColor(img, cv2.COLOR_BGR2HSV)
    new_points = []
    for color in my_colors.keys():
        lower = np.array(my_colors[color]['lower'])
        upper = np.array(my_colors[color]['upper'])
        mask = cv2.inRange(img_hsv, lower, upper)
        x, y = get_contours(mask)
        c = my_colors[color]['value']
        cv2.circle(img_result, (x,y), 10, c, cv2.FILLED)
        if x!=0 and y!=0:
            new_points.append([x,y,c])
        # cv2.imshow('image:'+color, mask)
    return new_points

def get_contours(img):
    contours, hierachy = cv2.findContours(img,
        cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_NONE)
    x,y,w,h = 0,0,0,0
    for cnt in contours:
        area = cv2.contourArea(cnt)
        print(area)
        if area>500:

```



```

        cv2.drawContours(img_result, cnt, -1, (0,0,255), 3)

        peri = cv2.arcLength(cnt, True)

        approx = cv2.approxPolyDP(cnt, 0.02*peri, True)

        x,y,w,h = cv2.boundingRect(approx)

    return x+w//2, y + h//10

def draw_on_canvas(my_points):
    for p in my_points:
        cv2.circle(img_result, (p[0], p[1]), 10, p[2], cv2.FILLED)

while True:

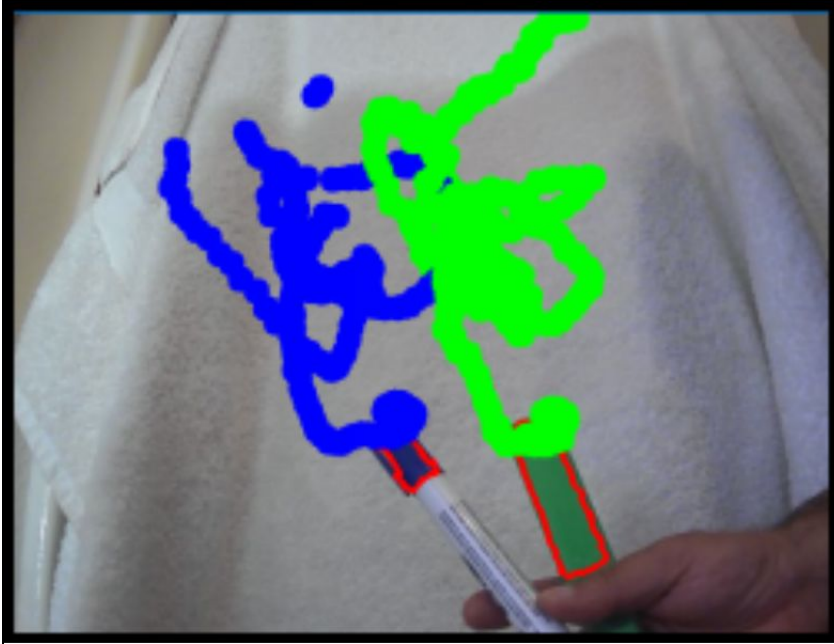
    # break at the end
    done, img = cap.read()
    if done == False:
        break

    img_result = img.copy()
    new_points = find_color(img, my_colors)
    if len(new_points)!=0:
        for newp in new_points:
            my_points.append(newp)
    if len(my_points)!=0:
        draw_on_canvas(my_points)
    cv2.imshow('Video', img_result)

    # break if 'esc' or 'q' is pressed
    key = cv2.waitKey(1)
    if key==27 or key==113:
        break

```

And here is the result:



A rather fun looking virtual painting application !

We have come to the end of the digital image processing section where the journey began with some 1's and 0's and extended to face recognition and virtual painting. We could have included other projects such as a document scanner or number plate detection. But these are left to the reader...

Number plate recognition

Detect movement

Object detection

## Web scraping

At this point, we have significant computing power. We also have methods of communicating between devices. And now we have knowledge of asynchronous methods for i/o connections which can cause bottlenecks.

Various systems push data into the ecosystem which is the web and the systems can extract the data. So for example, we can browse the web with a device, but we also have tools for automation.

We already looked at the `requests` module, which enabled the user to retrieve data as text, JSON or binary, which is good in many cases.

We can also connect via an Application Programming Interface (API) and collect the data a bit more efficiently.

However, there are powerful alternative methods with modules such as `selenium` and `beautiful soup`. We will take an overview of these methods and when we may or may not prefer one mechanism over the other and consider the “whoops, they redesigned” problem.

## Selenium

According to the official website “*selenium automates browsers. That’s it. What you do with that power is entirely up to you*”.

The first thing that we want to try is the equivalent of hello world in the selenium module. So let’s try to open a web page:

```
from selenium import webdriver
from selenium.webdriver.chrome.service import Service
from webdriver_manager.chrome import ChromeDriverManager
import time

service=Service(ChromeDriverManager().install())
driver = webdriver.Chrome(service=service)

url = "http://www.spacegoo.com/solar_system/"

driver.get(url)
time.sleep(30)
```

The first thing to note is that the selenium package requires more imports than other packages that we have seen before.

This is because the Selenium module needs to take into account all of the browsers that are available on the market (chrome, firefox, edge etc.) as well as all of the different versions that the browsers come in. So this in itself is a large task and hence a prime example of a use case for a module.

In fact the hello world example required 3 imports:

1. Selenium - the selenium module
2. Service - selects the chrome browser
3. ChromeDriverManager - gets latest version

The first is the webdriver tool itself. The second and third imports steps select *which browser* to automate and a convenient tool for the *latest version* of the selected browser.

Prior to the introduction of the manager tool (which was introduced in selenium version 4), the user was required to point to the driver file path on the pc and when an upgrade of the browser took place, the code needed to be updated to point to the new version, which was rather cumbersome given that many users tend to automatically upgrade to the latest version.

So the user effectively does an install the latest compatible browser at each run of the code and the result is ultimately assigned to the driver variable.

From this point, opening a browser with a website is simply a case of calling the same URL string that the user would have typed into the browser and then running the `driver.get()` function.

Once the user has arrived at a web page, it is often the case that they may want to do something in particular, like download a file, click on a link or press a button. And this is the automation part.

We will pause at this point because there are other automation processes that we should be considering before proceeding.

The hint for selenium is that its primary use case is automation and this is where user actions are required prior to accessing a web page.

<https://www.selenium.dev/>

## Beautiful soup

Beautiful Soup is a library that makes it easy to scrape information from web pages. It sits atop an HTML or XML parser, providing Pythonic idioms for iterating, searching, and modifying the parse tree.

This module is designed for quick turnaround projects like screen-scraping. In particular the following three features:

1. a few simple methods and Pythonic idioms for navigating, searching, and modifying a parse tree.
2. automatically converts incoming documents to Unicode<sup>[26]</sup> and outgoing documents to UTF-8. So the user does not need to think about encodings.
3. sits on top of popular Python parsers like lxml and html5lib.

We can get of to a quick start with the bs4 module:

```
from bs4 import BeautifulSoup
soup = BeautifulSoup("<p>Some<b>bad<i>HTML")
print(soup.prettify())
```

All the user needs to do is import the module and then parse any HTML string or document into the package.

And this is the output:

```
GussedAtParserWarning: No parser was explicitly specified, so I'm using the best available HTML parser for this system
("lxml"). This usually isn't a problem, but if you run this code on another system, or in a different virtual environment, it may use
a different parser and behave differently.
```

To get rid of this warning, pass the additional argument 'features="lxml"' to the BeautifulSoup constructor.

```
soup = BeautifulSoup("<p>Some<b>bad<i>HTML")
<html>
<body>
  <p>
    Some
  <b>
    bad
```

```
<i>  
  HTML  
</i>  
</b>  
</p>  
</body>  
</html>
```

We even get thrown the convenient error message of how to fix this error. We don't yet know what this error means other than it is to do with the structure of the document.

Like all code, we are expecting a properly structured (or well written) document, but this is not always the case, so a useful module like bs4 is designed to help with this issue.

We know how to download documents using the requests module in the HTTP request section of the book. So these modules usually come hand in hand [first the request, then the reading of the data].

Again, just like with selenium we will pause to consider the other automation processes.

## SQLite (Database)

The database section of the book could have been included into a number of the chapters and most notably the *data types* and the *data science* sections.

There are many different databases on the markets and these are broadly categorised into two sections.

- SQL
- No SQL

SQL and NoSQL differ in whether they are relational (SQL) or non-relational (NoSQL), whether their schemas are predefined or dynamic, how they scale, the type of data they include and whether they are more fit for multi-row transactions or unstructured data.

Python offers SQLite out of the box as one of the core modules and this was the case since version 2.5 (so prior to the existence of version 3).

*“SQLite is an in-process library that implements a **self-contained, serverless, zero-configuration, transactional SQL database engine**”.*

The terms in the description above are basically telling us that SQLite is a relatively simple database to work with. And it is the most widely deployed and used database engine which has found its way into every major operating system and also every web browser along with a host of major products and companies using it.

For this reason, it is suitable to use SQLite as the example case as it has the potential to provide use cases for both beginners and advanced users.

The key difference between SQLite and most other databases is that SQLite does not have a separate server process. SQLite reads and writes directly to ordinary disk files.

In general, databases are faster than file systems and SQLite claims to be approximately 35% faster than a standard file. One main reason for this is that the open() and close() system calls are invoked only once for databases.

Most SQL database engines use static, rigid typing. So each column is effectively given a type at the outset.

These are the types that SQLite offers:

- NULL. The value is a NULL value.
- INTEGER. The value is a signed<sup>[27]</sup> integer.
- REAL. The value is a floating point value.
- TEXT. The value is a text string.
- BLOB. The value is a blob of data, stored exactly as it was input.

So the Python language has good familiarity with the types with the introduction of the “blob”.

SQLite does not offer the boolean data type, but this is handled with 1 (True) & 0 (False).

SQLite also does not have a Date and Time Datatype, so handles this as the **Text**, **Real** or **Integer** types. For example the integer as Unix time represents the number of seconds since 1970-01-01 00:00:00 UTC.



We have discussed a lot, so let's get into the code. We make a database like this:

```
import sqlite3

conn = sqlite3.connect('employee.db') # makes a database file
c = conn.cursor() # make a cursor

# start running sql commands
c.execute(""" CREATE TABLE employees (
    first text,
    last text,
    pay integer
) """)

conn.commit() # commit data to the database & close
conn.close()
```

With the sqlite3 module we are able to create a database with just a few lines of code. In fact, we could create an **in memory** database by replacing “employee.db” with the string :memory: but for the purpose of the example we created a file.

The process for most of the database functions in python (and other languages) is similar.

1. Connect to a database
2. Execute SQL commands (to receive or send data).
3. close the connection.

In the example, “employee.db” contains some data on employees salary. The first two columns for the names are **text** and the next column for salary was an **integer**.

We can data to the database like this:

```
c.execute('insert into employees VALUES ("alice", "apple", 50000)')
c.execute('insert into employees VALUES ("bob", "banana", 75000)')
```

And view the results in vscode with the SQLite extension.

SQL ▾			< 1 / 1 > 1 - 2 of 2		
first	last	pay			
alice	apple	50000			
bob	banana	75000			

We can access the same statement in python with the `select` statement:

```
c.execute('SELECT * FROM employees WHERE last="apple"')
result = c.fetchall()
print(result)
```

Which gives this:

```
[('alice', 'apple', 50000)]
```

Which is what we were expecting. There are in fact 3 fetch commands:

- `fetchone()` - returns the next row
- `fetchmany(n)` - return the next n rows
- `fetchall()` - return all rows

So, at this point we are able to set up, populate and retrieve (query) data with a database in python in just a few lines of code.

The advantage of accessing the database via code is that we can programmatically add data from lists, dicts or even classes or variables like this:

```
employee_3 = {'first':'clive', 'last':'banana', 'pay':60000}
c.execute('insert into employees VALUES (:first, :last, :pay)',
        employee_3)
```

Note that we avoid using string formatting which is vulnerable to SQL injection attacks and instead pass the object (we used a dict) directly into the database with the colon notation representing the database columns.

The same method as above can be extended to queries:

```
c.execute('SELECT * FROM employees WHERE last=:last',
          {'last':'banana'})
result = c.fetchall()
print(result)
```

Which now returns a list of tuples because there were two employees of the same name (banana).

```
[('bob', 'banana', 75000), ('clive', 'banana', 60000)]
```

So, again we are confident in the result.

One useful feature of databases in python is that we can pull all the data into a pandas dataframe in a similar way that we were able to for text and csv files using the `read_sql()` method in just one line of code.

For the purpose of completeness, we will cover one other aspect of python with databases which is using functions and also context managers (the `with` statement) to conveniently operate on the database.

Insert a new employee:

```
def insert_emp(emp:dict):
    with conn:
        c.execute('insert into employees VALUES (:first, :last, :pay)',
                  emp)
```

Retrieve an employee:

```
def get_emps_by_name(lastname):
    c.execute('SELECT * FROM employees WHERE last=:last',
              {'last':lastname})
    return c.fetchall()
```

Update an employees salary:

```
def update_pay(emp:dict, new_pay:int):
    with conn:
        c.execute("""UPDATE employees SET pay = :pay
                    WHERE first = :first AND last = :last""",
                  {'first':emp.first, 'last':emp.last, 'pay':new_pay})
```

Remove an employee

```
def remove_emp(emp:dict):
    with conn:
```

```
c.execute("DELETE from employees  
WHERE first = :first AND last = :last",  
{'first':emp.first, 'last':emp.last})
```

So we now have convenient functions to operate on the database in an easy format.

And we can call the functions to perform operation in a user friendly way:

```
employee_01 = {'first':'dave', 'last':'cucumber', 'pay':100000}  
employee_02 = {'first':'elle', 'last':'Dates', 'pay':125000}  
  
insert_emp(employee_01) # add employee  
insert_emp(employee_02) # add employee  
  
update_pay(employee_02, 95000) # update pay  
remove_emp(employee_01) # remove employee  
  
emps = get_emps_by_name('banana') # get data  
print(emps)
```

And with this we are now able to add, remove and update the pay of salaries.

SQL ▼			< 1 / 1 > 1 - 4 of 4		
first	last	pay			
alice	apple	50000			
bob	banana	75000			
clive	banana	60000			
elle	Dates	95000			

As well as being able to make queries.

There are a whole host of other things that one can do with SQLite, so it is left as an exercise for the reader to explore this.

## Automation processes

Getting data is important. Infact, aside from setting up the environment it is the first task that a process usually has whether that is in data science or any other field.

There are some key aspect of data acquisition that one needs to consider:

- Quantity of data
- Speed (latency) of transfer
- Complexity of the data
- Easy of access
- Evolution of the source

The above are some headline considerations, but can be further broken down which is why it was prudent to pause on the selenium and beautiful soup modules.

### **Quantity:**

Some pieces of data might only be a few lines. For example the price of a particular item in a supermarket. This might be in the form of a web query where we would use the requests module.

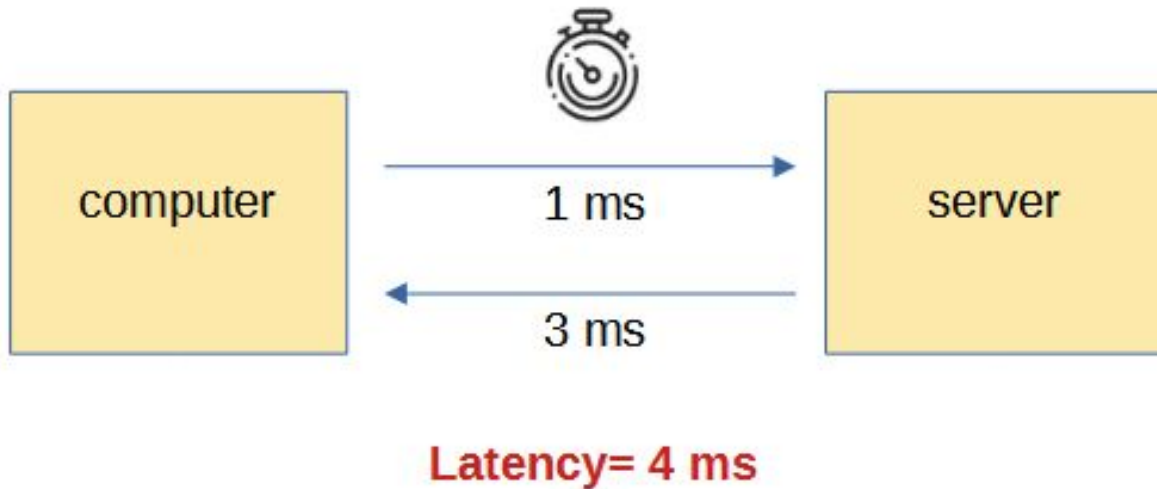
We might also have light data in the form of the price of a particular stock, but this would be continuously changing throughout the course of the day, which would be more suited to a websocket.

### **Speed:**

Aside from the volume of data requested, various sources take different lengths of time to be acquired into the system. For example reading a csv file via the pandas or csv modules is a relatively fast process compared to an http request via the web.

If a user was to make many http requests, then they would need to consider asynchronous methods (such as asyncio, threading or multiprocessing) such that the requests do not block each other.

We call this “speed” the **latency** of the network which involves the round trip of the data request and receiving the data pursuant to the request.



So reading a file is low latency whereas reading a web page is high latency.

#### **complexity:**

Next we need to consider the complexity of the data. This will encompass factors like which format does the data come in, for example text, csv or json and also how structured in the data.

A flat csv file will read very quickly into a pandas dataframe, whereas a json file is larger, but may contain richer content such as many nested levels.

We also need to consider encoding of the data and if omitted it would break or mean something different in the selenium and bs4 modules than we may otherwise have expected.

#### **Ease of access:**

This aspect might seem trivial, but there are a whole host of features which prevent simple data transfer. This could be as simple as access to a database or a password or link or forwarded page in an html document where the

user logs into one page and is sent elsewhere to a second or third page to get the data that they need.

Sometimes, the method of access will change when the system is updated. Like an upgraded API.

The other issue is that some web hosts are averse to openly sharing data in a machine readable format or be scraped by bots and have measures in place to prevent this. The user would then have to resort to alternative measures such as a headless browser.

### **Evolution of the source:**

The internet, databases and other sources such as the likes of ccd arrays (ie. web cameras) are continually evolving. Most prominently the evolution and format of web pages. It is quite common to create a crawler that works very well for a significant period of time, but then breaks when the formatting of the web page changes.

This is because a typical html document is made up from elements and with a refactoring of the website a number of the terms will change, just like a a refactoring of code might prompt the coder to modify their variable names even if it is for a simple reason like clarity.

Nevertheless, if the scraping code is searching an HTML page for an element or xPath<sup>[28]</sup> that no longer exists and relies on this data, then the scraping code will be compromised.

And this is the curse of the “whoops, they redesigned” problem.

### **The optimal tool:**

Finally, with a more complete knowledge of the products available, let's explore various data sources and the appropriate Python tool for completing the task.

<b>Content type</b>	<b>method</b>
files	<ul style="list-style-type: none"><li>• Read and write with</li></ul>



	context managers. <ul style="list-style-type: none"> <li>• Read csv, pandas.read_csv</li> <li>• Read json</li> </ul>
database	Database tools like SQLite
Web page	<ul style="list-style-type: none"> <li>• <a href="#">With basic text</a>: requests, beautiful soup</li> <li>• <a href="#">Tables</a>: pandas read_html()</li> <li>• <a href="#">with data file</a>: requests, selenium</li> <li>• <a href="#">With clicks / navigation</a>: selenium</li> </ul>
Peripherals	treat as files

Note that there are many ways to skin the cat, but the above table provides some useful guidelines.

## Others

Other ideas that are not yet complete.

### My first problem

This was actually probably not my first ever problem, but it was the first time that I was stumped enough to have the need to go to the stack to ask a question for Python as previously I had been using Node.JS which is another popular scripting language (and one which uses curly braces , semicolons and `//` for comments).

The process of asking a good question, which we can define as one in which the community will give you up votes instead of down votes is a rather lengthy process.

There is actually a specific format that was alluded to earlier which we will adhere to time and time again. This is the **Minimal, Reproducible Example**.

At this point I am concerned enough to ask the question in such a way that the community can see that I have:

- researched the issue sufficiently
- provided enough code that others can replicate the problem quickly
- made an attempt at what I consider to be a solution
- clearly state what I do and don't understand
- very specific in what the desired output should be

If the user has not done these basic things, then they risk being downvoted.

So, let's get back to the point, my problem was this:

HTTPError: Forbidden

I get `HTTPError: Forbidden` error when trying to read from a website that requires feedback response in the form of radio buttons. How to resolve this ???

An example url link that causes the error is commented out: `url = 'http://www.londonstockexchange.com/exchange/news/market-news/market-news-detail/WWH/13673174.html'`

This was something very specific and annoying at the time. Whilst I was able to use a specific library (in this case the `requests` module). I was unable to navigate beyond the first press of a button on the website.

I was trying to do something rather simple, let the code navigate a web page for me, but was getting stuck early on and the web page in question required the user to click a radio box button and if they did not, then they would get this `Forbidden` error.

I provided the code that I had done:

```
import urllib.request as req
import re

from bs4 import BeautifulSoup

# get all text from websites.
url = 'https://www.bbc.co.uk/sport/football'
#url = 'http://www.londonstockexchange.com/exchange/news/market-news/market-news-de
html = req.urlopen(url).read()
soup = BeautifulSoup(html)

# kill all script and style elements
for script in soup(["script", "style"]):
    script.extract()    # rip it out

# get text
text = soup.get_text()

# break into lines and remove leading and trailing space on each
lines = (line.strip() for line in text.splitlines())
# break multi-headlines into a line each
chunks = (phrase.strip() for line in lines for phrase in line.split(" "))
# drop blank lines
text = '\n'.join(chunk for chunk in chunks if chunk)

print(text)
```

This particular question got 79 views and nobody posted any answers whatsoever.

However, one person posted a comment.

You can try using selenium to simulate the radio button selection and button click. I now look back, some 4 years later, to find yet another user who was much better than I was (with double my stack score) who had pointed me to a relatively popular package that I had not come across before.

Basically, what he was saying was “you are using a spanner to open a tin, why don’t you try using a tin opener?”

The point being that I was using completely the wrong tool for the job.

At this point, I could go into the specifics of the selenium package that he was referring to, but this would defeat the purpose. The principle of coding, like all other aspects in life, is using the right or at least the best tools for the job.

A good coder will generally know what these are, just like a carpenter will have specific tools for slicing and dicing wood or an electrician for fixing wires.

The answer somehow required a package (of the many thousands available) that was capable of quickly navigating the web browser and until I had this particular tool, I would always struggle.

## My first answer

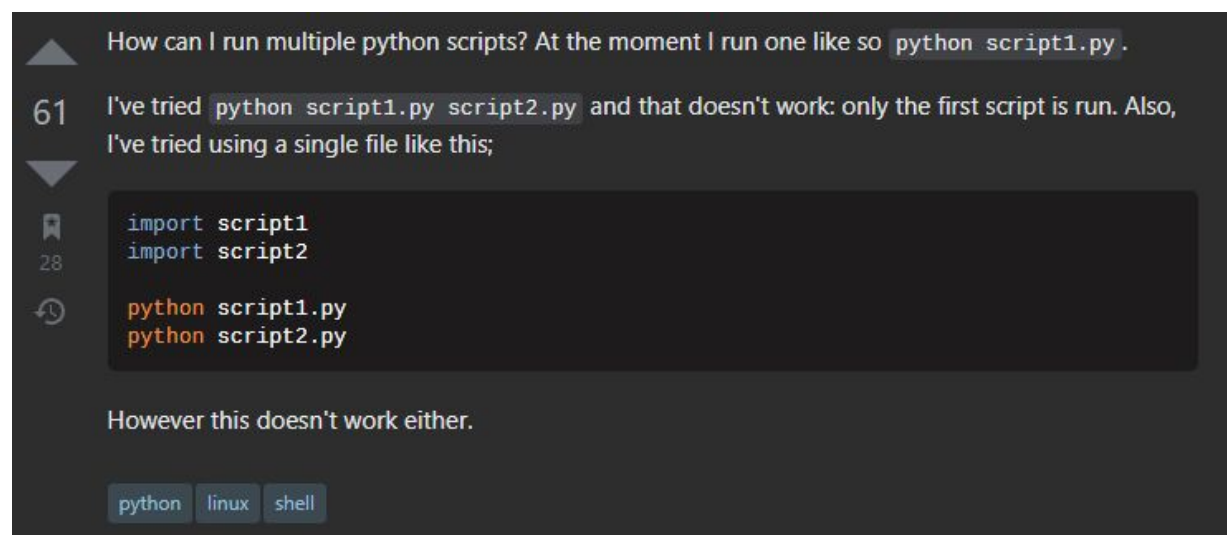
Before I go into this, I note that I have answered 3 times as many questions as I have asked. I don't know what this means other than I might be a net contributor into the system as I continually struggle with problems and would have thought that the balance would have actually been the other way around.

So maybe I don't ask enough questions?

Nevertheless, let's look at the question that I answered.

The title was: Run multiple python scripts concurrently

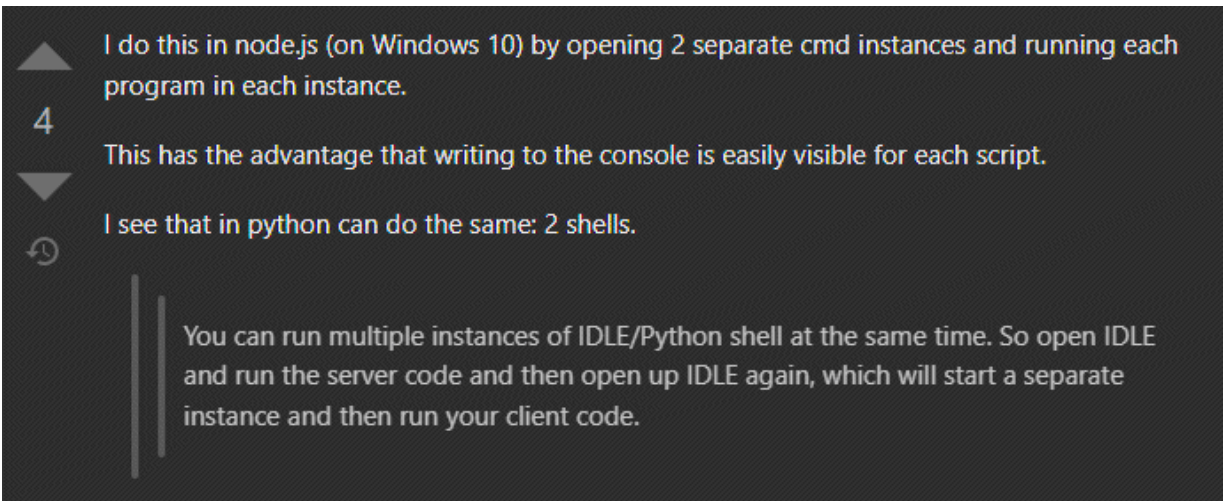
And the question was this:



We can see that it had 61 up votes, so clearly a respectable question in the sense that many people probably faced the same issue.

And over the years, there have been several answers posted.

I had stumbled upon the same question some 3 years after it was posted, obviously trying to do the same thing. In fact I faced this problem or at least a very similar issue with Node.js which is another scripting language so I posted a solution to how I would do this in node and how the same rules transfer over to python.



This was one of many solutions and probably not even the best of the solutions, nevertheless sufficient to solve a problem for a few users with the same issue in a simple manner.

Again, the point here is not the technicalities of the answer, but that as a coder we can draw techniques from other similar languages. **It is actually the general rules that we are looking first at as opposed to the specific details.**

The second point to note, which was far more interesting, is that the user wants to run multiple code concurrently. A massive issue in the coding world which crops up literally everywhere.

## 100 prisoners

So this problem does not yet exist on the stack, but there is a great video on youtube of the same. Here we will go through the problem.

There are 100 prisoners and each prisoner has to go into a room. The room contains 100 boxes. And each box contains a different number. Each prisoner can pick up to 50 boxes. If the prisoner picks their own number, then it is a success. If all 100 prisoners get successes, then they are all set free. If even one of the prisoners fails (more than 50 boxes), then they all die.

What is the probability that the 100 prisoners survive ?

## Classes and mutability

<https://stackoverflow.com/questions/73537529/unexpected-behavior-with-python-list-of-dictionaries-when-adding-new-key-to-them/73538560#73538560>

Here was the original question:

*“I have encountered unexpected behaviour from my python interpreter during a project. I'm aware that, in certain situations, python uses the reference to an object, not the object itself. However, I cannot explain why this problem arises in this piece of code since the elements of the list seem to be different instances with different object ids. Whenever I add a new {key:value} pair to the dictionary, all other dictionaries in that list get updated.”*

```
class Node():
    def __init__(self, name, neighbors=dict()):
        self.name = name
        self.neighbors = neighbors # a dict {}

    def add_neighbor(self, neighbor, value=0):
        self.neighbors[str(neighbor)] = value

if __name__ == "__main__":
    n_1 = Node(name='node_1')
    n_1.add_neighbor('node_2', value=5)
    n_2 = Node(name='node_2')

    node_list = [n_1, n_2]
    for node in node_list:
        print(id(node), node.name, node.neighbors)
```

The result is this:



```
2129022374520 node_1 {'node_2': 5}
2129022374576 node_2 {'node_2': 5}
```

So why did this happen and how do we fix this problem?

The default value for “neighbors” is a single dictionary that is shared across class instances. This means that every instance points to the same dict. It is an inherent part of the mutability of lists and dicts (the memory address does not change when the value changes). This is a useful feature when used properly, but when overlooked or misunderstood could cause problems.

So how do we rectify this?

One method would be to just type-hint of a dict in place of assigning a dict. Remember, a type-hint in python is exactly that. This then means that the user could assign their dicts in each instance when creating (instantiating) the object.

```
class Node():
    def __init__(self, name, neighbors:dict): # hint
        self.name = name
        self.neighbors = neighbors # a dict {}

    def add_neighbor(self, neighbor, value=0):
        self.neighbors[str(neighbor)] = value

if __name__ == "__main__":
    n_1 = Node(name='node_1', neighbors={}) # dict a
    n_1.add_neighbor('node_2', value=5)
    n_2 = Node(name='node_2', neighbors={}) # dict b

    node_list = [n_1, n_2]
    for node in node_list:
        print(id(node), node.name, node.neighbors)
```

- 
- [1] Pandas is derived from the words panel data. It has nothing to do with panda bears.
- [2] The PIL library provides fairly powerful image processing capabilities.
- [3] Random Access Memory, RAM, is essentially short term memory where data is stored as the processor needs it.
- [4] Hard disk: Seagate BarraCuda Q5 NVMe SSD  
RAM: AMD Ryzen 3 4300U
- [5] We can install pandas with `pip install pandas` from a command prompt.
- [6] Vectorisation works best for Numeric code, but less well for strings.
- [7] This is known as the ndarray.
- [8] a contiguous block of pointers where each pointer references a Python object.
- [9] A crude look at the website appears to offer in excess of 1000 basic types.
- [10] Unpacking is another convenient feature in python.
- `a,b = np.polyfit(...)` gets both the gradient and intercept variables in one line of code.
- [11] The iris flower dataset: The data set consists of 50 samples from each of three species of Iris (Iris setosa, Iris virginica and Iris versicolor).
- [12] KNN stands for  $K^{\text{th}}$  Nearest Neighbour.
- [13] JSON stands for **J**ava**S**cript **O**bject **N**otation.
- [14] A drawdown is a sell off in the market to a specified amount.
- [15] Uniform Resource Locator, which is effectively the unique web address.
- [16] A race condition occurs when two or more threads can access shared data and they try to change it at the same time. Because the thread scheduling algorithm can swap between threads at any time, you don't know the order in which the threads will attempt to access the shared data.
- [17] The Python Global Interpreter Lock (GIL) is a lock that allows only one thread to hold the control of the Python interpreter. So only one thread can be in a state of execution at any point in

time. The GIL has no impact on single-threaded programs, but it can be a performance bottleneck in asynchronous code.

[18] Tkinter originates from Tk which is a free and open-source, cross-platform widget toolkit that provides a library of basic elements of GUI widgets for building a graphical user interface (GUI) in many programming languages. The first version was released in 1991.

[19] A callback function is *a function passed into another function as an argument*, which is then invoked inside the outer function to complete some kind of routine or action.

[20] Originally coined in the 17th century by René Descartes as a derogatory term and regarded as fictitious or useless, the concept gained wide acceptance following the work of Leonhard Euler (in the 18th century) and Augustin-Louis Cauchy and Carl Friedrich Gauss (in the early 19th century).

[21] Historically, the geometric representation of a complex number as a point in the plane was important because it made the whole idea of a complex number more acceptable.

[22] Data sampling is important as it can yield different results. The sampling depends on how frequently the data is inspected relative to the quantity of the data available..

[23] Scaremongering is one format of rumour propagation. The 'fear' in each node (person) is likely to activate a transmission of data in a proportion relative to the particular nodes 'fear;' threshold. We can refer to this as triggering the activation function.

[24] Moore's law is the observation that the number of transistors in a dense integrated circuit (IC) doubles about every two years. Moore's law is an observation and projection of a historical trend. Rather than a law of physics, it is an empirical relationship linked to gains from experience in production.

[25] The method is named after its creator John F Canny is a popular edge detection algorithm which uses image gradients.

[26] Everyone in the world should be able to use their own language on phones and computers. The Unicode Standard allows for consistent encoding, representation, and handling of text expressed in most of the world's writing systems.

[27] A signed integer is a 32-bit datum that encodes an integer in the range [-2147483648 to 2147483647]. The signed integer is represented in two's complement notation (1,2,3..8 bytes).

[28] XPath stands for XML Path Language. It uses a non-XML syntax to provide a flexible way of addressing (pointing to) different parts of an XML document.