DOWNING COLLEGE CAMBRIDGE
Summer School

Applied Mathematics
Online Course

Keenan J. A. Down

Summer 2024

1 Applied Mathematics at University

1.1 Welcome!

Welcome to the Downing College Summer Program 2024! I'm going to be teaching you all some applied mathematics over the next couple of weeks, and I'm super excited that you're here to join me.

Over the next two weeks we're going to try and develop some mathematical background to help you study on other courses over the course of the summer school. Many of the subjects you might want to study might make use of this material, and we've worked together to help provide a basic mathematical foundation for the other subjects.

Over the course of the next two weeks we're going to look at a broad variety of topics, from programming, to data analysis, to mathematical modelling. Here's a rough outline of the structure of the rest of this course:

1. Applied Mathematics at University
2. Introduction to Python
3. A Bit More Python
4. Distributions and Data
5. Hypothesis Testing
6. Exploratory Data Analysis
7. Simulation and Hacker Statistics
8. A very brief introduction to Machine Learning

As you can see, we're going to lean heavily into the kinds of skills that you might find useful moving towards university study. Quite a lot of the course is focused on helping you develop a small amount of fluency with Python, a leading programming language. Most research these days is performed using code, and python is one of the most celebrated and widely used programming languages available today. It's also free to use.

Starting at the beginning, we'll explore some foundational topics when working with data, developing some intuitions for working with the code. We'll then move towards statistics from the programming perspective, before looking at some really modern ideas in data. Along the way we'll explore a few additional topics to help supplement your study in the other courses.

Let's get started!

1.2 How can mathematics help me at university?

Being mathematically literate is incredibly important in the modern world, and it's never too late to develop some more mathematical skills. This is especially true if your main area of study at university isn't highly mathematical - that is, if you do something besides STEM. Having good data literacy gives us additional skills to understand lots of confusing modern phenomena - the AI boom comes to mind. What exactly *is* a neural network anyway, and why do they suddenly seem to be everywhere?

Even if not for yourself, learning to apply mathematical skills, reason with data, and explore trends will give you lots of new tools for exploring your own subject.

- Mar's project - exploring data in psych and art? politics/economics?

1.3 Mathematics at university

Learning mathematical skills at university is very different to the kinds of mathematics you might have done in school thus far. At school, lots of mathematics is very formulaic - learning a method to apply to a very specific kind of problem. Often this goes so far as to learning the approach for every different variation on a given class of problems (you might know, for example, three different ways to solve a quadratic equation).

At university, this kind of structure is turned a bit onto its head. Mathematical study might start with quite a lot of structure, but over the course of your programme, you're likely to notice that questions become much more open-ended, freeform and exploratory. This is true more or less immediately for mathematics students - if you study mathematics as your main subject at university, the main goal is to develop a feeling for the art of **problem solving**. That is, you might be given many slightly ambiguous questions, and often be asked to determine an answer without any fixed method in particular. This is very classic for maths students - luckily the rest of us are spared this toil until later into the course!

Depending on what area you go into at university, you're likely to learn mathematics quite differently. For people studying science, technology or engineering subjects, especially the very mathematically heavy ones such as Physics and CompSci, you're likely to encounter a first-year/first-term mathematics course. In that course you're likely to learn about some **analysis** - that is, a slightly more rigorous introduction to *calculus*, the mathematical study of change, and some statistics and mechanics relevant to your subject. Regardless of what you do, you're unlikely to be able to escape studying calculus, and if you're lucky, you might even get to do **multivariable calculus** (lots of fun).

If you choose to do mathematics, or follow a more pure-mathematical thread through your university life, you're likely to study mathematics in a very rigorous way. You might see, for instance, mathematics classes being taught as a sort of collection of facts or statements, which might look like this:

Theorem 1 (Pythagoras' Theorem)**.**
Given a right-angled triangle with sides a, b, and hypotenuse c, we have

$$a^2 + b^2 = c^2$$

Proof. The proof is left to the reader. □

In more formal mathematical education, you'll often find that you're presented with these statements and definitions and asked to learn them and their proofs (that is, a series of steps justifying why they are correct). The thinking is, by studying results and proofs carefully and often enough, you'll learn how to construct interesting results and produce correct proofs when you explore the subject more on your own.

In more science/engineering subjects, you'll much more likely find that the "proofs" are omitted, taking only interesting results to benefit the other material that you're going to learn.

In a science discipline, you're likely to encounter a lot more statistics along the way, often aided by statistical software like SPSS or RStudio. It is quite common in modern scientific research to forego statistical software for statistical testing using a programming language (like Python or R). Python, in particular, is very heavily employed in many disciplines these days and is a great starting language for almost anyone to learn.

Figure 1: IBM SPSS, the R programming language, and the Python programming language.

For example, if you want to demonstrate that a drug is actually making a significant intervention against a certain disease (and actually has an effect), you're going to want to be able to compare the results between the control group and the intervention group. Performing a valid statistical test will tell you whether or not the difference between the two groups actually exists, or if it's more likely that it's being caused by just random noise. For this reason, don't be surprised if you take a psychology class at university and suddenly find yourself in a statistics class!

Overall, you're likely to find that mathematics at university feels much more open-ended than in school. You won't be given a method to solve every possible version of a problem that you're possibly going to see. Instead, it's far more important to develop a generalist and broad-ranging understanding of the problems itself and reasonable mathematical approaches you might take, so that you are able to reason for yourself and solve unseen problems.

A large part of this means that, very often, you're going to find yourself in a situation where you don't know how to solve a problem. You might realise you don't know something as well as you thought, or you simply have no idea why the numbers aren't coming out right. This is part of the experience when you come to using mathematics at university, and, most importantly, you shouldn't panic when this happens! Just like when look at a page of a book in a foreign language, you don't panic because you don't understand, you shouldn't panic when you come across problems you don't immediately know how to solve - instead, you take a deep breath, accept it might take some time to wrap your head around it, and get to work learning the language. Before long, a lot of it gets a lot easier - but that experience will never completely go away.

1.4 Mathematics for your career

Surprisingly, mathematical skills are more in-demand than ever, yet so many people don't feel comfortable with their numerical skills. As mentioned, lots of people experience a sort of *maths panic*, where they freeze, decide they're not a "maths person" and then move on while trying to avoid it for the rest of their life.

This is a real shame - mathematics is really the study of patterns, and humans can't help but see patterns in everything! So, unsurprisingly, when you're able to spot patterns in your work and use the mathematics to really state the pattern carefully, this can be really valuable to your business and your career.

This can be relatively simple things, like creating a model for predicting how much stock a business still has, to complicated data-driven models for difficult forecasting problems. Individuals who, at the very least, aren't *afraid* of using mathematics to think about business problems tend to be highly valuable, especially if they're also good at communicating that information to a business audience (which might be rather general).

Increasingly these days we find AI and Machine Learning (ML) are becoming ever-present in our tech. Everything from the prediction engine powering your phone's autocorrect, to your spotify recommendations, to enormous, gargantuan language models like GPT-4, Claude 3 or Gemini Ultra. What these models do is, essentially, allow us to capture patterns in data (be that text, similar listening

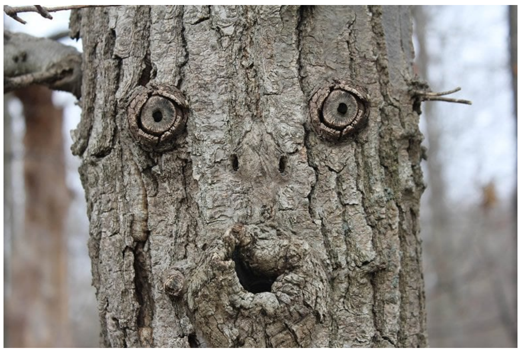


Figure 2: **Pareidolia** is the natural tendency to perceive patterns in stimuli where the pattern does not exist. Humans are hard-wired to search for meaning in things, even if it isn't really there.

habits, or all of the text in the internet) and make use of those patterns to reap enormous economic benefits.

If you can get a foothold in the work of ML, it can take your career very far very quickly, as many organisations these days are looking to expand their use of AI to maximise growth. As such, lots of routes have opened in recent times to enable individuals to move into tech, even if they've worked in different areas previously. For example, there are now a collection of courses called "AI Bootcamps", where individuals, usually having achieved at least a BSc before, go to learn (usually online) about machine learning models and data science for 6 weeks or so. Many of these courses also offer career guidance at the end of the programme, giving even more help for individuals to get into a data-centric career (which, by the way, tend to be very lucrative - the average data scientist, according to Glassdoor at the time of writing, earns 52,000 pounds in the UK).

Most of this course is going to be directed at learning about how to think of mathematics as a tool for studying data. That might be data from research, or more broadly for business. As such, we're going to explore some of the basic ideas behind this data-science approach. Regardless of what you take away, a lot of the content you learn here could be useful for your future career (if you're willing to apply it!).

1.5 Projects across disciplines

Lastly, it's worth mentioning that there are many areas where we can use applied mathematics to create interesting artistic works. For example, if you haven't heard of DALLE-3 [7] or Midjourney (or even Sora [8], which was announced two weeks ago at the time of writing), then you've been missing out on some of the most bizarre generative technology that humans have ever created. For example, here are some pictures that DALLE-3 generated when I provided it with a prompt.

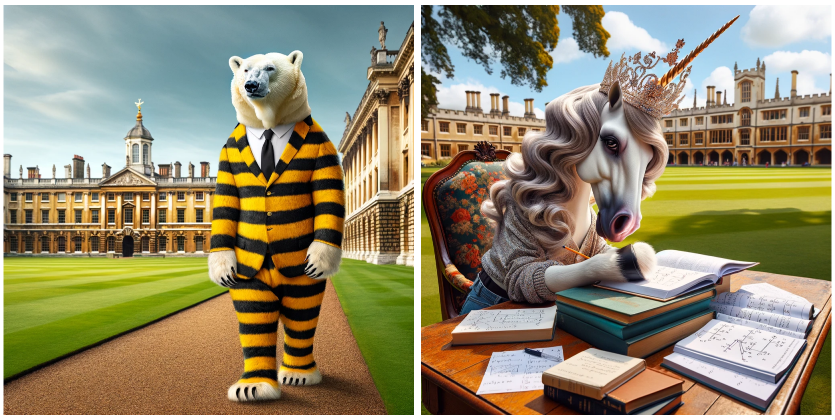


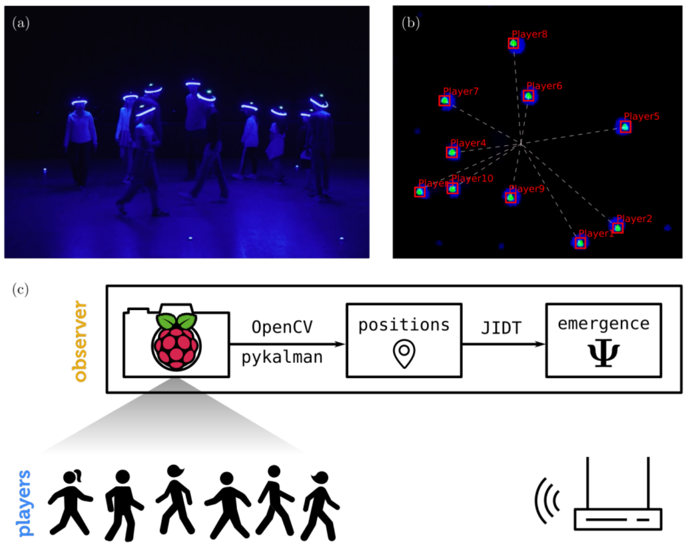
Figure 3: "Please make a picture of a polar bear wearing a suit, walking through Downing College, Cambridge. His suit is inspired by the striped pattern of a bee."
"Could you now make a picture of a fashionable unicorn wearing a crown studying applied mathematics at the Downing College Summer Programme?"

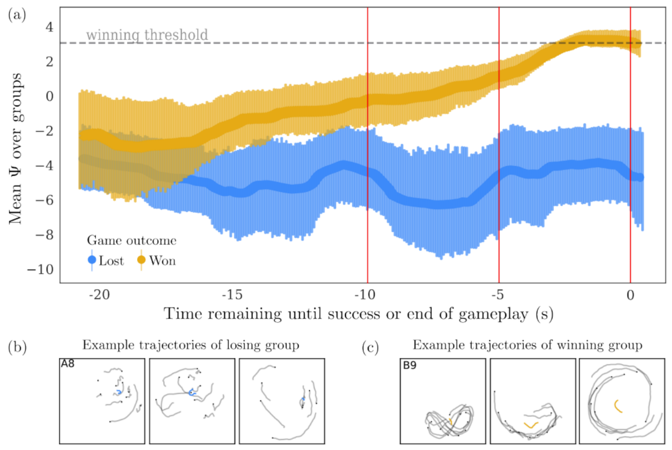
While these particular examples are somewhat mortifying, models like DALLE-3 and Midjourney are incredibly powerful and can be helpful for producing graphics in scenarios where the need for human-produced art is relaxed somewhat, like in presentations or background art on a website.

Two more artistic projects that have come out of work taking place in the Cambridge Consciousness and Cognition Lab (my main lab) are **Synch.live** [12] and **Latent Ecologies of the Mind** [9]:

Synch.live

Synch.Live was an artistic and scientific exploration of how humans feel more connected when they exhibit emergent behaviour. Individuals were placed in a dark room wearing flashing hats and given the task, as a group, to get the flashing of the hats to sync up. To achieve this, they used a measure Ψ reflecting emergence [11]. As this number increased, the synchronicity of the flashing also increased, giving the players a goal of acting as an emergent group (without explicitly mentioning this goal).





Successful groups developed ”flocking” motions, where the centroid (centre point) of the group mass was more predictive of the group’s behaviour than the parts added together [12].

Latent Ecologies of the Mind

In this project, electroencephalogram (EEG) was used to record brain activity from two people listening to ‘L’ile Neu’ by Hikaru Hayashi [9]. The signals were then decomposed.

After some processing, the two signals are then convolved to create a single signal, representative of an *ecotone* between two minds (a sort of transient state between biological systems).

From here, Lempel-Ziv Complexity (LZc), a standard measure of complexity which can be applied to brain signals to take a rough measure of someone’s consciousness level, was applied to the convolved signal.

By considering how the LZc varied with time and with delay between the two signals, a 2D surface was created to represent possible complexities. This was then fed to a diffusion model to produce a visualisation of their joint *ecology* [9].

The results are visually very interesting!

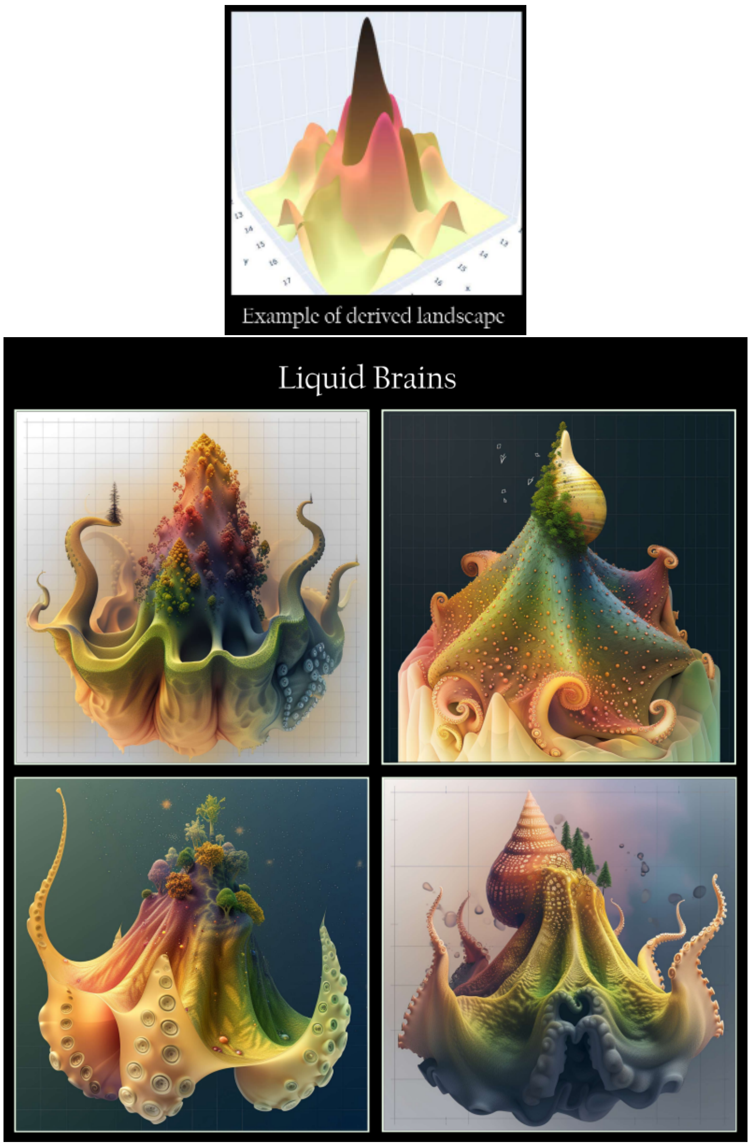


Figure 4: A derived 'landscape' of complexities across time and delay, and the output of the diffusion model when provided the landscape as a starting point.

As you can see, there are lots of new and emerging avenues where a little knowledge of applied mathematics can take you in research, art, science, and technology. Many of these new avenues are building on new technologies such as large language models and diffusion models.

In this course we're going to learn a bit more about just how beautiful data can be. I hope you enjoy the course!

2 Introduction to Python

In the hopes of helping you to get hands-on with data, we're going to learn a little bit of programming. Python is a high-level object oriented programming language. High-level means that it's great for abstracting difficult tasks away from system-level functionality, so you don't need to know very much about the machine where the code is being run. *Object-oriented programming* is also about abstraction; an object-oriented programming language will be structured around various objects or data structures. With Python, it's especially easy to build structures and objects which you can later use to better manage and structure your code. Furthermore, python is free to use and widely used in industry - it's probably one of the most in-demand programming languages you could learn, so it's a great place to start.

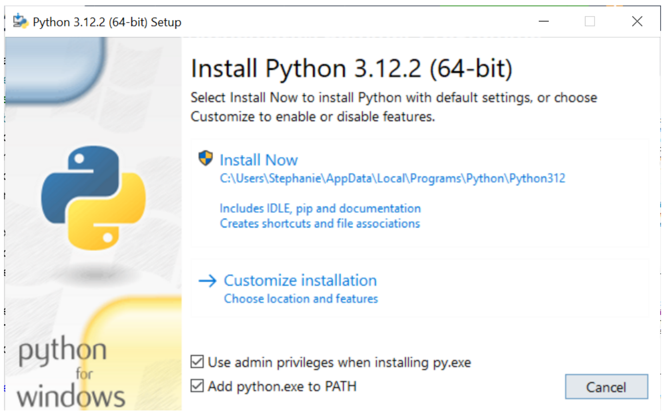
As we spoke about yesterday, learning to program is one of the most useful skills you can have in the modern world. Almost all of the technology that you encounter every day is built with code, and python, with its relatively mild learning curve and intuitive syntax, is very widespread. Without further ado, let's get started.

2.1 Getting Started

We'll start by installing Python onto your device. Go to

`https://www.python.org/`

and go to **Downloads**, and install the current version of python onto your device (3.12.2 or later). There'll be lots of help on how to do this online, and it'll depend slightly on your operating system. During the installation, make sure that "add python.exe to PATH" is ticked, as this will make it easier to use from the command line.


Python 3.12.2 (64-bit) Setup

Install Python 3.12.2 (64-bit)

Select Install Now to install Python with default settings, or choose Customize to enable or disable features.

Install Now
C:\Users\Stephanie\AppData\Local\Programs\Python\Python312

Includes IDLE, pip and documentation
Creates shortcuts and file associations

Customize installation
Choose location and features

☑ Use admin privileges when installing py.exe
☑ Add python.exe to PATH

Some UNIX distributions (MacOS and Linux) have a version of Python pre-installed, but you might want to follow the above steps to just make sure it's up to date anyway.

There are a few different ways you can run python. We're going to heavily prefer the last of these three options, but it's worth knowing regardless.

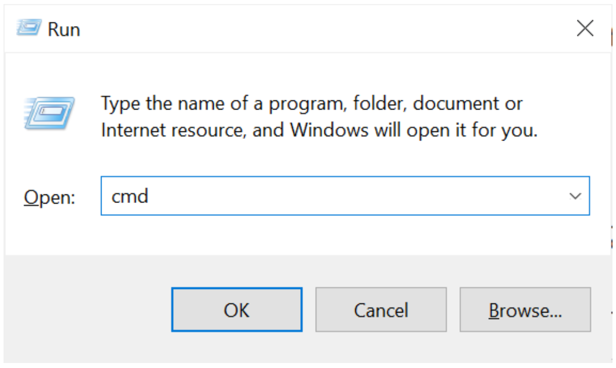
2.1.1 At the command line

When you install python, the installer normally adds python *to the PATH* when you use a command line tool like **Command Prompt** (Windows) or **Terminal** (MacOS, Linux). This means that you can use the python interpreter (python.exe, the executable file that runs code) by using `python` at the command line.

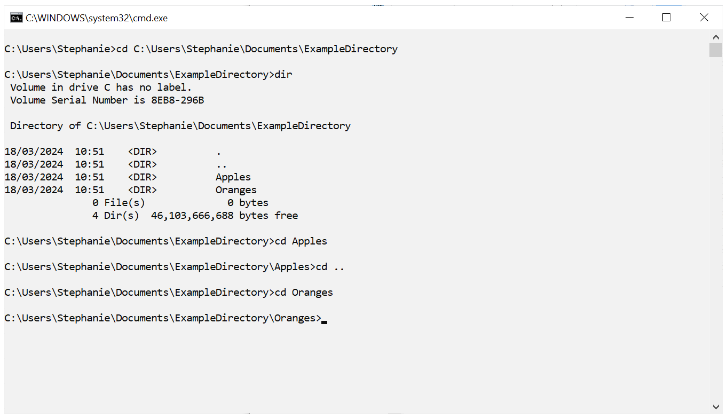
On Windows you can open the command prompt quickly by running `windowskey+r` Typing in `cmd` and hitting enter will then open up an instance of the Command Prompt.

Alternatively you can find it in the start menu or by searching in the task bar.

You'll notice that once you've opened it, you will have a blinking cursor in a line which contains the name of a directory. Here it was "`C:\Users\Stephanie`". This is the **current working directory**,


Run
Type the name of a program, folder, document or Internet resource, and Windows will open it for you.
Open:
cmd
OK
Cancel
Browse...

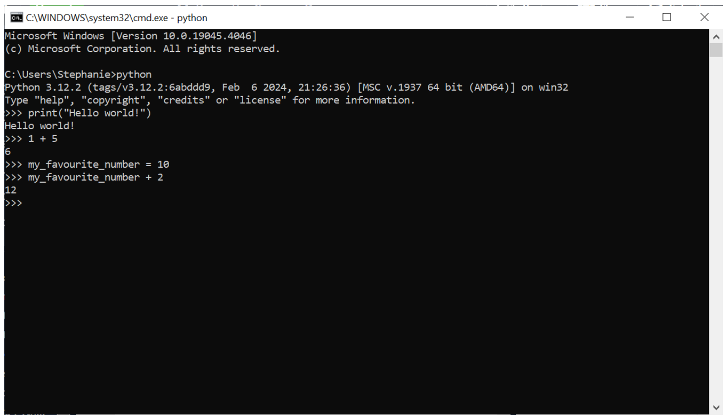
and reflects the "location" you're currently accessing using the command prompt. The current working directory will affect what your code is able to see. If you need to change it, you can use `cd` to navigate to a folder, or `cd ..` to navigate up one level (out of the current folder). You can use `dir` in Command Prompt or `ls` on MacOS/Linux (UNIX systems) to see the files in the current working directory.



C:\WINDOWS\system32\cmd.exe

```
C:\Users\Stephanie>cd C:\Users\Stephanie\Documents\ExampleDirectory

C:\Users\Stephanie\Documents\ExampleDirectory>dir
 Volume in drive C has no label.
 Volume Serial Number is 8EB8-296B

 Directory of C:\Users\Stephanie\Documents\ExampleDirectory

18/03/2024  10:51    <DIR>          .
18/03/2024  10:51    <DIR>          ..
18/03/2024  10:51    <DIR>          Apples
18/03/2024  10:51    <DIR>          Oranges
               0 File(s)              0 bytes
               4 Dir(s)  46,103,666,688 bytes free

C:\Users\Stephanie\Documents\ExampleDirectory>cd Apples

C:\Users\Stephanie\Documents\ExampleDirectory\Apples>cd ..

C:\Users\Stephanie\Documents\ExampleDirectory>cd Oranges

C:\Users\Stephanie\Documents\ExampleDirectory\Oranges>_
```

This might be useful later. For now, you can simply type `python` in any directory to start the python interpreter in interactive mode. This will prompt you to enter python code (after the > > >), and you can explore basic python code by sending one line of the code at a time.



C:\WINDOWS\system32\cmd.exe - python

```
Microsoft Windows [Version 10.0.19045.4046]
(c) Microsoft Corporation. All rights reserved.

C:\Users\Stephanie>python
Python 3.12.2 (tags/v3.12.2:6abddd9, Feb  6 2024, 21:26:36) [MSC v.1937 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> print("Hello world!")
Hello world!
>>> 1 + 5
6
>>> my_favourite_number = 10
>>> my_favourite_number + 2
12
>>>
```

Here, the first line of code we sent was `print("Hello world!")`. The `print` function takes a `string` as input (i.e. a piece of text, usually provided with "" or ''), and prints it out to the command line. Naturally `print` is very useful for following the behaviour of complicated python code.

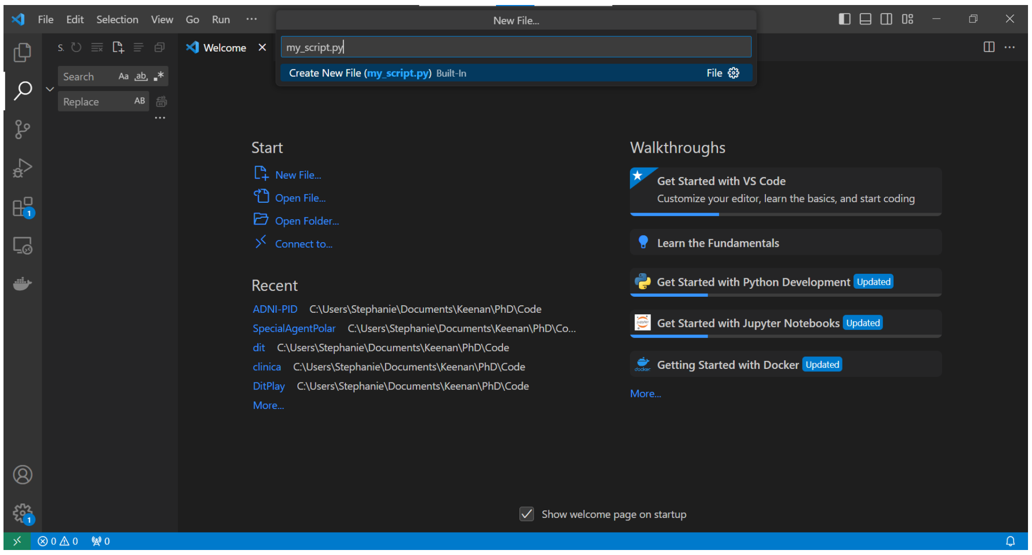
You'll notice that Python can, by default, do the standard mathematical operations (+, -, * and /), and in interactive mode, it usually automatically prints the output of the last line of code.

Most of the time it won't be particularly neat or helpful to run code at the command line in interactive mode, as you won't be able to save your code easily, or make changes quickly to what you've written. One other way of using the command line for running python is via **scripts**.

2.1.2 Scripting and running the code

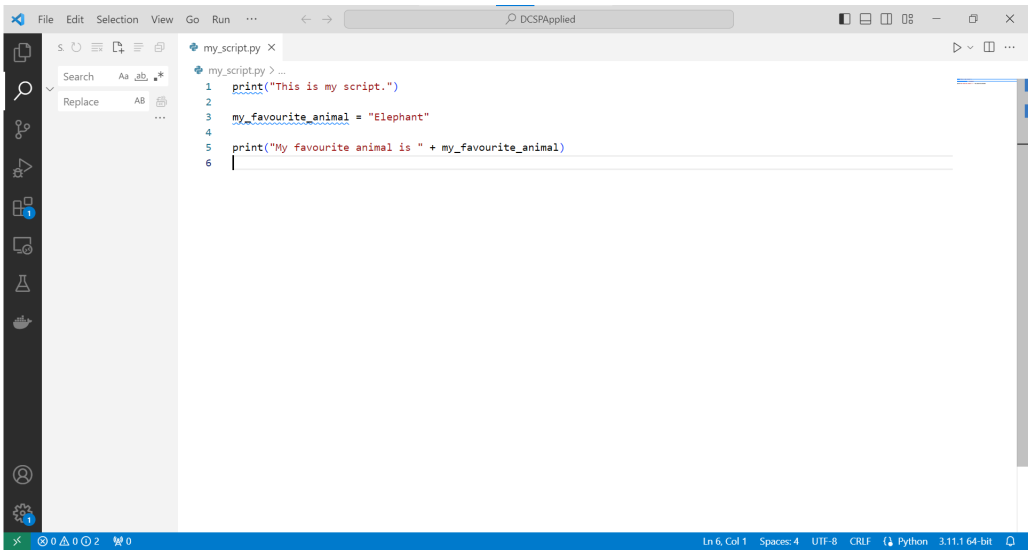
Often times it's much more convenient to collect the code you write together into a file called a **script**, which is simply a text document containing the code which you can run later.

To do this, you'll usually want to use a code editor or IDE (integrated development environment) to give you somewhere to write it. While in principle it's possible to write code in Notepad or TextEdit, developer tools often make writing code much easier, as they often provide suggestions for corrections or completions for standard functions (often like auto-correct, but for code). My current default choice on windows is **VSCode** (Visual Studio Code), which you can install free at `https://code.visualstudio.com/`. It's also available for MacOS and Linux.



File Edit Selection View Go Run ···
New File...
Welcome
my_script.py
Create New File (my_script.py) Built-In
File
Search
Replace
Start
New File...
Open File...
Open Folder...
Connect to...
Recent
ADNI-PID C:\Users\Stephanie\Documents\Keenan\PhD\Code
SpecialAgentPolar C:\Users\Stephanie\Documents\Keenan\PhD\Co...
dit C:\Users\Stephanie\Documents\Keenan\PhD\Code
clinica C:\Users\Stephanie\Documents\Keenan\PhD\Code
DitPlay C:\Users\Stephanie\Documents\Keenan\PhD\Code
More...
Walkthroughs
Get Started with VS Code
Customize your editor, learn the basics, and start coding
Learn the Fundamentals
Get Started with Python Development Updated
Get Started with Jupyter Notebooks Updated
Getting Started with Docker Updated
More...
Show welcome page on startup

Here I've clicked on `New File...` and given the new file the name `my_script.py`. Pressing enter, you might also get a dialogue box asking you to choose where to place the file. Ensure that the file name ends in `.py`, as this is the extension for python scripts.



File Edit Selection View Go Run ···
DCSPApplied
my_script.py
my_script.py > ...
Ln 6, Col 1 Spaces: 4 UTF-8 CRLF Python 3.11.1 64-bit

```python
print("This is my script.")

my_favourite_animal = "Elephant"

print("My favourite animal is " + my_favourite_animal)

```

Here we've written a short script (and saved it using `crtl + s`). You'll notice that VSCode actually has put some blue squiggly lines under some of my code. This is just because it doesn't conform to normal programming standards (which is more for people who work at software companies like Google etc. who need to keep very standard code). For our purposes this doesn't matter!

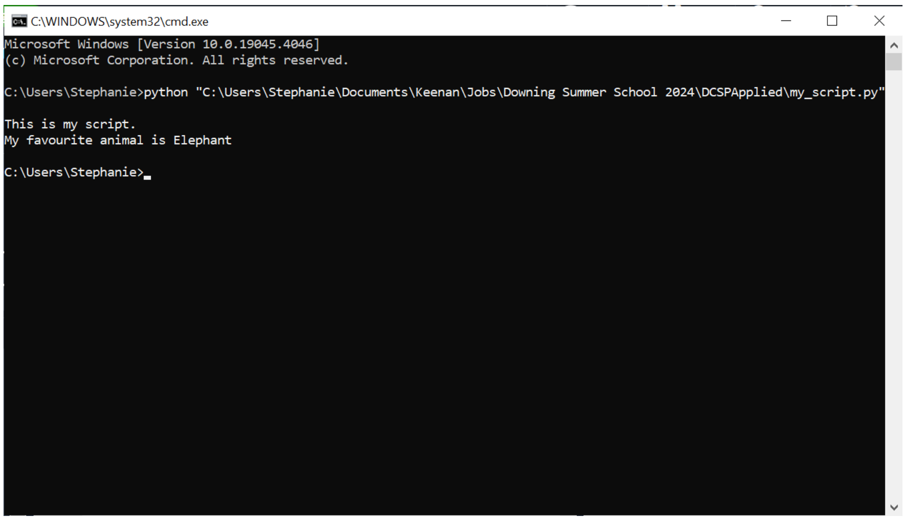
The script will print out two lines. The first one will say "This is my script.", and then the second one will say "My favourite animal is Elephant". Note that in Python, we can concatenate two strings using $+$.

While it is possible to run the code directly inside of VSCode (see run and debug on the left hand side), we're going to try running the code form the command prompt.

Open your command line app once again. Find the complete filename of your code (with all parent directories) "C:/Users/.../my_script.py". We're then going to run

```
python "<your file directory>"
```

As you can see, this runs the code in our script line by line, producing the output we were expecting[1].



```
Microsoft Windows [Version 10.0.19045.4046]
(c) Microsoft Corporation. All rights reserved.

C:\Users\Stephanie>python "C:\Users\Stephanie\Documents\Keenan\Jobs\Downing Summer School 2024\DCSPApplied\my_script.py"

This is my script.
My favourite animal is Elephant

C:\Users\Stephanie>
```

Running python scripts this way can be incredibly useful, especially if you need to perform calculations on a remote server or autonomously. Much of my research takes place on a High Performance Computing cluster - essentially a supercomputer - and as they have no graphical interfaces, all code is run like this.

This is great, but what if we want to go back, change around some things, try the script again, and so on? It's not always incredibly effective to jump back and forth from the script to the terminal. Moreover, it makes it quite hard to convey interesting results with text - and it's not easy to convey a story with our code. For that we'll turn to our last, mostly preferred, method.

2.1.3 Running code in Jupyter Notebooks

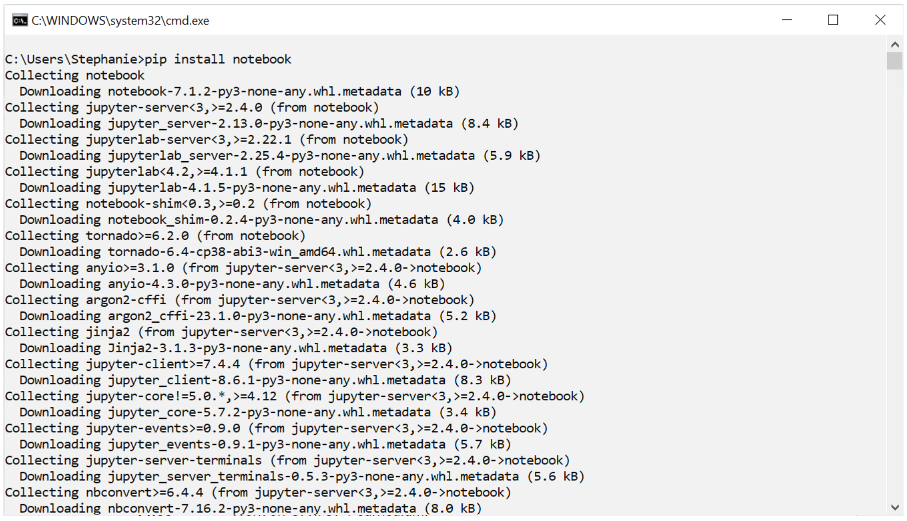
A **Jupyter Notebook** is a file which is able to do two things: run code and display normal text. For that reason, they're often very useful when you want to dip in and out of your code, or want to be able to take notes about what you're doing.

They're also a relatively common format online for sharing work on small challenge projects. For example, if you work on a challenge project on kaggle.com, you might create a Jupyter Notebook to walk people through your code.

Jupyter notebooks are quite easy to set up. When you installed Python, you also installed a command utility called `pip`, which stands for "Preferred Installer Program", or "PIP Installs Packages" if you're feeling meta. A **package** is a collection of scripts called **modules** which you can import and re-use. For example, if you want to use python to work with data, you're likely to use the package `pandas`, which we'll meet later.

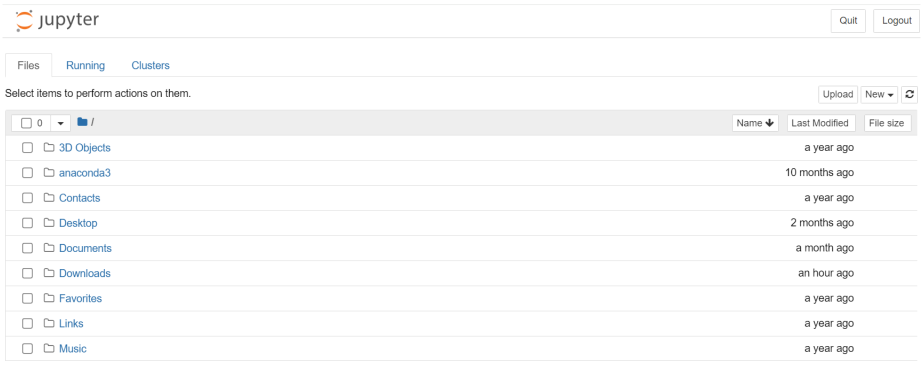
[1] If you're wondering who Stephanie is, I have no idea - the university lent me this laptop when I was having some problems with my own!

We'll come back to importing modules later. For now we're going to use `pip` to install Jupyter Notebook. To do this, go to the command line and perform `pip install notebook`. When you do this, make a note of the directory you see at the command line. Ours was ""



```
C:\Users\Stephanie>pip install notebook
Collecting notebook
  Downloading notebook-7.1.2-py3-none-any.whl.metadata (10 kB)
Collecting jupyter-server<3,>=2.4.0 (from notebook)
  Downloading jupyter_server-2.13.0-py3-none-any.whl.metadata (8.4 kB)
Collecting jupyterlab-server<3,>=2.22.1 (from notebook)
  Downloading jupyterlab_server-2.25.4-py3-none-any.whl.metadata (5.9 kB)
Collecting jupyterlab<4.2,>=4.1.1 (from notebook)
  Downloading jupyterlab-4.1.5-py3-none-any.whl.metadata (15 kB)
Collecting notebook-shim<0.3,>=0.2 (from notebook)
  Downloading notebook_shim-0.2.4-py3-none-any.whl.metadata (4.0 kB)
Collecting tornado>=6.2.0 (from notebook)
  Downloading tornado-6.4-cp38-abi3-win_amd64.whl.metadata (2.6 kB)
Collecting anyio>=3.1.0 (from jupyter-server<3,>=2.4.0->notebook)
  Downloading anyio-4.3.0-py3-none-any.whl.metadata (4.6 kB)
Collecting argon2-cffi (from jupyter-server<3,>=2.4.0->notebook)
  Downloading argon2_cffi-23.1.0-py3-none-any.whl.metadata (5.2 kB)
Collecting jinja2 (from jupyter-server<3,>=2.4.0->notebook)
  Downloading Jinja2-3.1.3-py3-none-any.whl.metadata (3.3 kB)
Collecting jupyter-client>=7.4.4 (from jupyter-server<3,>=2.4.0->notebook)
  Downloading jupyter_client-8.6.1-py3-none-any.whl.metadata (8.3 kB)
Collecting jupyter-core!=5.0.*,>=4.12 (from jupyter-server<3,>=2.4.0->notebook)
  Downloading jupyter_core-5.7.2-py3-none-any.whl.metadata (3.4 kB)
Collecting jupyter-events>=0.9.0 (from jupyter-server<3,>=2.4.0->notebook)
  Downloading jupyter_events-0.9.1-py3-none-any.whl.metadata (5.7 kB)
Collecting jupyter-server-terminals (from jupyter-server<3,>=2.4.0->notebook)
  Downloading jupyter_server_terminals-0.5.3-py3-none-any.whl.metadata (5.6 kB)
Collecting nbconvert>=6.4.4 (from jupyter-server<3,>=2.4.0->notebook)
  Downloading nbconvert-7.16.2-py3-none-any.whl.metadata (8.0 kB)
```

Once you've finished installing it, you can open Jupyter Notebook by running `jupyter notebook` at the command line. Once you do this, your default web browser will open and you should be greeted with a page like this:

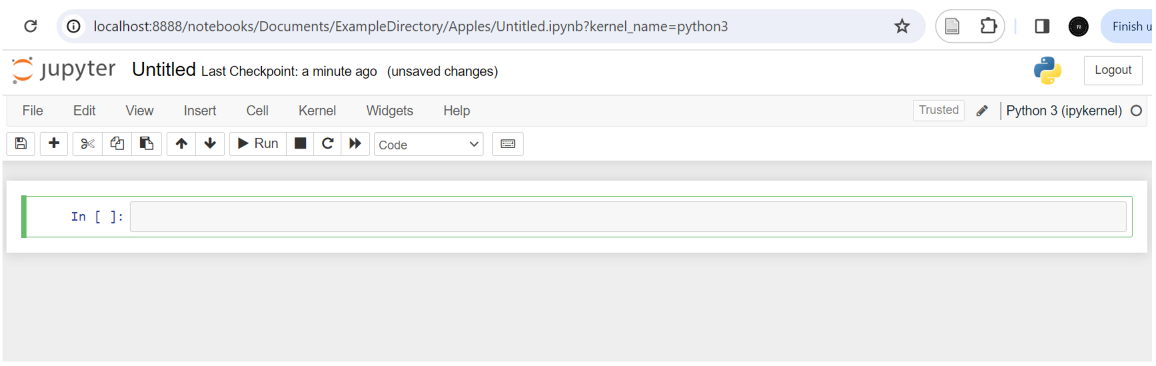


Select items to perform actions on them.

Name	Last Modified
3D Objects	a year ago
anaconda3	10 months ago
Contacts	a year ago
Desktop	2 months ago
Documents	a month ago
Downloads	an hour ago
Favorites	a year ago
Links	a year ago
Music	a year ago

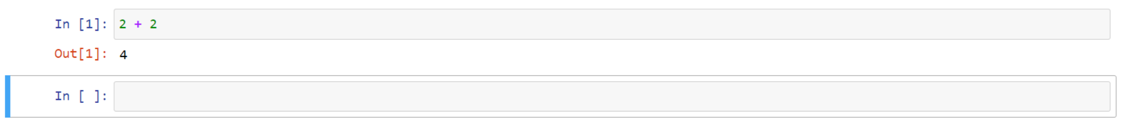
What you'll see is a navigation window for all of the files inside of the current working directory where you ran `jupyter notebook`. You won't be able to access files higher than your working directory, so set this accordingly (see subsection 2.1.1).

Now you can find a desired directory, go to `New` and then `IPython 3` (it might also just say Python or something similar). This should make a new jupyter notebook that looks like this:


localhost:8888/notebooks/Documents/ExampleDirectory/Apples/Untitled.ipynb?kernel_name=python3
jupyter
Untitled Last Checkpoint: a minute ago (unsaved changes)
Logout
File
Edit
View
Insert
Cell
Kernel
Widgets
Help
Trusted
Python 3 (ipykernel)
Run
Code
In [ ]:

You can change the title by clicking on the `Untitled` at the top of the screen, and you can save the file under `File > Save and Checkpoint`.

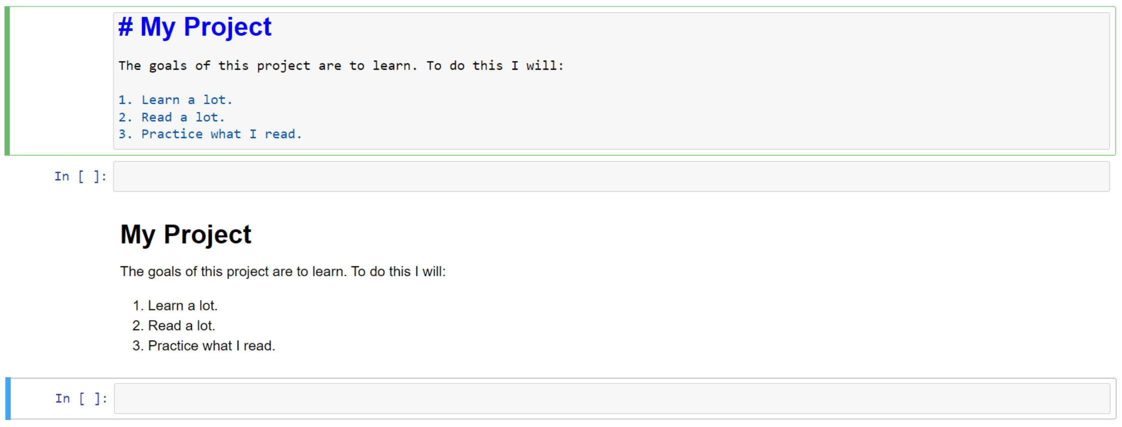
What you see is a box, called a **cell**, where you can write code. To execute the code that's in the cell, you can press `shift + enter`. Any output from the code will appear at the bottom of that cell, and you'll move into a new cell.



```
In [1]: 2 + 2
Out[1]: 4
In [ ]:
```

To make more cells, you can either `shift + enter` after the last cell, or you can click on a cell (it'll turn blue - green means you're editing a cell) and press `a` to create a new cell *above* the previous one, or `b` to create a new cell below. You can click on a cell and press `d` twice to delete a cell.

In addition to being able to run code, however, you can also write text. Making a new cell, selecting it, and pressing `m` will make it a **Markup** cell, which means that it will output text. **Markup** is a markup language which makes it easy to make nice looking documents. For example, in markup, this is what the following cell looks like when it is "run":



```
# My Project

The goals of this project are to learn. To do this I will:

1. Learn a lot.
2. Read a lot.
3. Practice what I read.

In [ ]:
```

My Project

The goals of this project are to learn. To do this I will:

1. Learn a lot.
2. Read a lot.
3. Practice what I read.

```
In [ ]:
```

Hopefully you can see how Jupyter Notebooks can be useful for organising ideas or experimenting with new pieces of code. They're especially good for data analysis, as they allow you to keep all of your "storytelling" in one single place - your code, your analyses and your insights. We'll make lots of use of Jupyter Notebooks for the rest of this course and you'll learn to use them to analyse data.

If you have issues installing any of the software, a quick google search should help you - there should be lots of material online for how to install python and Jupyter notebook. If all else fails, ChatGPT might be able to give you some ideas!

2.2 Basics in Python

2.2.1 Variables and Comments

Variables

Let's get into actually using the language now. In most programming language, **variables** are short-hand representations of data. They contain data and store it so that it can be used again (or sometimes changed) later. Let's define some variables.

```
my_name = "Keenan"
my_age = 26
my_height = 1.83

print("Hi, my name is " + my_name + "!")

>> Hi, my name is Keenan!
```

In the code above we defined three variables, **my_name**, **my_age** and **my_height**, all of which store different kinds of information. We can use that information again later. For example, if we have a variable which contains **text**, then we can use `print` to produce that variable as output. When we have two strings, we can use + to concatenate them, which gave us the output above.

Comments

A really good habit when programming is to use **comments**. These are small pieces of text which help developers and other people using the code to understand what it does. In Python, we make in-line **comments** using #, which means that anything on the line after the # won't form part of the code, but can be read to help you (and others) understand what the code does. It's good practice to use lots of comments - enough so that someone who hasn't read your code before can understand what it does easily. Good comments speed the process of programming up enormously.

For example:

```
# This will not be read by the program.
text = "This will be read by the program."

x = 10
# x = 15

print(text)
print(x)

>> This will be read by the program.
>> 10
```

2.2.2 Classes and Types

As we mentioned before, Python is an **object oriented** programming language, which means that it places emphasis on storing information in different *types* of variables. These different structures are called their **type** or **class** (the difference between the two is massively beyond the scope of this course!)

One instance of a class is called an **object**. For example, "this" and "that" are both **instances** of the **string** class - they are string **objects**. Don't worry too much about understanding all of these words right now. We won't be building our own classes in this course, so you don't need to worry too much about it right now.

If you want to find out what class a variable has, you can use the function `type`, which will return the class of a given variable.

```
# Define my name.
my_name = "Keenan"

# Print the type.
type(my_name)

>> str
```

Here you see that **my_name** has the string class, denoted by **str**. Let's talk about a few different built-in kinds of objects you're going to encounter.

String (str)

Strings are used to contain text data, and you can specify them by using single quotes or double quotes. If you want them to span over multiple lines, you can use three quotation marks `"""` or `'''` to signify the start and the end.

```
my_name = "Keenan"

my_story = """I am a spy from the future,
sent back in time to teach
at the Downing College
Summer Programme."""

print(my_name)
print(my_story)
>> Keenan
>> I am a spy from the future,
   sent back in time to teach
   at the Downing College
   Summer Programme.
```

String objects are **immutable**, which means you can't change the underlying data after you create the variable. Instead it will just be overwritten.

Integer (int)

Integers are just that - whole numbers. If you specify a whole number as a variable, its type will automatically be assigned to `int`. Naturally you can add, subtract, multiply (*) and divide (/). You can also do modular arithmetic with %, which will give the remainder after division. Integers are immutable.

Float (float)

When you want to work with numbers which aren't whole numbers, the variable will be saved as a float. For example, 23.12 or 46.2 are both floats. The word *float* is short for *floating-point number*, meaning that there is a decimal place. Floats are immutable.

List (list)

Python starts to get a lot more interesting when you introduce lists. A list, naturally, is a list of things. For example, here's a list:

```
# Define a list
my_list = ["cat", 10, 3.141, ["dog"]]
```

You can get the elements of a list at the n-th entry by doing `my_list[n]`. Note that Python uses **zero-indexing**, so to get the first element of the list, you'll do `my_list[0]`, and so on.

Lists can consist of any class, or even a mix of classes. It's not uncommon to create a list of lists. Here, the fourth entry $n = 3$ will be a list itself.

```
print(my_list[0])
print(str(my_list[1]))
print(my_list[3][0])

>> cat
>> 10
>> dog
```

To create a new list of a series of consecutive elements, you can use a **slice**.

```
my_new_list = my_list[1:3]
my_new_list
>> [10, 3.141]
```

Note that this is also zero indexed and won't include the second index. This is to prevent double-counting so that `my_list[1:2]` and `my_list[2:3]` don't overlap.

List are **mutable**, and you can change a single entry by doing `my_list[n] = "blah"` or something similar to change the $n-1$-th entry.

Function (function)

You've used a few functions so far. These are like mini snippets of code which perform a certain task. For example, `print` is a function, but there are many others. You can also define your own functions, but we won't cover this in this course (even though it's really interesting!)

Dictionary (dict)

Dictionaries are a great way of organising data. Instead of trying to remember the order of pieces of data in a list, you can use dictionaries to order data really well. They're built out of two parts; **keys** and **values**. When given a certain *key*, the dictionary will return the corresponding *value*.

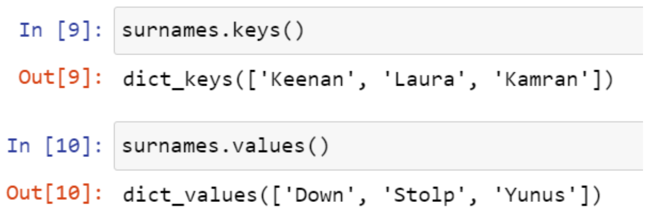
Here's an example:

```
# Define the dictionary.
surnames = {
    "Keenan": "Down",
    "Laura": "Stolp",
    "Kamran": "Yunus"
}

# Print out Keenan's surname.
print(surnames["Keenan"])
>> Down
```

Dictionaries are incredibly convenient. You can't use every kind of class as a key (e.g. you cannot use mutable objects as keys), but they're still incredibly useful for organising data.

Note that they're delimited by curly brackets , with keys separated from their values by a colon :, with commas separating each key-value pair.

To see what the keys and the values of a certain dictionary are, you can append `.keys()` or `.values()` - these are two special functions attached to an object (functions attached to objects in this way are called **methods**).



```
In [9]: surnames.keys()

Out[9]: dict_keys(['Keenan', 'Laura', 'Kamran'])

In [10]: surnames.values()

Out[10]: dict_values(['Down', 'Stolp', 'Yunus'])
```

These methods return special classes, but you can convert them back to lists by using `list(surnames.keys())` if you need.

Dictionaries are mutable. They're incredibly versatile.

Tuple (tuple)

Tuples are a bit like lists but are immutable, so they can be used as dictionary keys. You can also use them to pass arguments to a function, if it takes multiple arguments.

Set (set)

Sets are unordered lists, which, much like the mathematical notion of a set, do not contain duplicate elements. You can specify the elements in a set by delimiting them with curly brackets, like dictionaries, but without the colons.

```
# Specify my set.
my_set = {"one", "two", "three", "three"}

print(my_set)
>> {'one', 'two', 'three'}
```

Sets are mutable. There are a few associated methods like `.add()`, `.remove()`, `.union()`, `.intersection()`, `.difference()`, which take elements or sets and allow you to perform usual set-theoretic operations.

Boolean (bool)

The last class we're going to mention for now is the Boolean. A Boolean stores whether or not something is `True` or `False`.

Later, when we mention *control flow* (changing which code runs depending on different conditions), we'll use boolean values a lot.

To define a Boolean in Python, you should write True or False (with the capitalisation included).

Basic logic gates can be accessed with &, | and `not()`. For example:

```
x = True
y = False
print(x & y)
print(x | y)
print(not(x))
>> False
>> True
>> False
```

You can also write `and` and `or` instead of & and | to make your code more readable, if you like. Booleans are not mutable.

These classes will keep coming up again and again, so it's good to get familiar with them. Open up Jupyter Notebook and have a play around!

3 A Bit More Python

Over the course of this session we're going to try and get a little bit more familiar with Python, so that we can start thinking about how to apply it to data. To do this, we'll first have to look at two key ideas: *control flow* and *functions*.

3.1 Control Flow

3.1.1 Logic

Control flow is about how we direct Python to execute different parts of code. For example, we might want to test whether or not a certain condition is true before proceeding, or change the behaviour of the program depending on context.

The **Boolean** type is incredibly useful for managing control flow. Consider the following two lines of code, and notice the difference:

```
my_number = 2

print( my_number == 2 )
>> True
```

The first line is using the assignment operator =. In Python, this means that you store the data after = in the variable which comes before. So this first line saves the variable **my_number** with value 2 and type `int`.

The second line looks similar but is actually performing a different operation. When we use ==, we're testing for **equality**, meaning that this whole statement will return as either `True` or `False` depending on whether or not the data stored in **my_number** is exactly equal to 2. In this case it is.

We saw previously that we can use `&` and | to reflect **and** and **or** respectively. For example, if I want to check that several conditions are true, I could try the following:

```
x = 1
y = 2
z = 3
print( (x == 1) & (y == 2) )
print( (x + y == z) | (y == 5) )
>> True
>> True
```

Note that you should always use brackets between the operators, otherwise Python might try and compute the expression differently (not using brackets here would give you two False outputs, actually).

If we want to invert a Boolean (flip `True` to `False` and vice versa), we can use `not()`:

```
print( not(True) )
print( not(False) )
>> False
>> True
```

There are some other logic operations that will often be useful. In addition to == we have `!=`, the **not equal** operator. This is equivalent to `not(a == b)`, but is often a quicker shorthand. Of course, when we're working with floating point numbers or integers, we also have the following logical operators:

```
>       # Greater than
>=      # Greater than or equal to
<       # Less than
<=      # Less than or equal to
```

You'll probably need these operations a lot!

3.1.2 "If" blocks

One of the most important control flow tools is the `if` clause. This checks that a certain statement is true, and if it is, runs a select window of code. Let's see how it looks on an example involving two cats:

```
# Specify Gerald's age.
age_gerald = 17

# If Gerald is old, print "Gerald is old".
if age_gerald >= 10:
    print("Gerald is old!")

# Specify Belle's age.
age_belle = 9

# If Belle is old, print "Belle is old".
if age_belle >= 10:
    print("Belle is old!")

>> Gerald is old!
>>
```

You can see here that the program only ran the code that printed "Gerald is old", because his age satisfied the conditions. The program didn't print anything out about Belle, because the statement wasn't satisfied.

What if we want to test several conditions and return the first piece of code where the condition is true? Then we can use `elif` and `else`. The `elif` token is short for **else if**. This code will run if all of the statements in the `if` block before it did not already execute and a given statement is true. At the end of the chain, we have the `else` token, which describes what should happen in all other cases. Here's the syntax:

```
if STATEMENT_1 :
    CODE TO RUN IF STATEMENT_1 IS TRUE
elif STATEMENT_2 :
    CODE TO RUN IF STATEMENT_1 IS FALSE AND STATEMENT_2 IS TRUE
elif STATEMENT_3 :
    CODE TO RUN IF STATEMENTS 1 AND 2 ARE FALSE AND STATEMENT_3 IS TRUE
else:
    CODE TO RUN IN ALL OTHER CASES
```

This is a great way of managing several complex logical cases while keeping your code relatively short. You can also chain `if` several times.

Two important reminders - don't forget the colon! This will cause your code to throw an error, as it won't know where the loop begins. Also, notice that there is an indentation inside of repeated blocks of code. This indentation is important to get right in python - not getting the spacing right will mean it doesn't always understand where a block of code begins or ends, and you're likely to get a `SyntaxError`.

3.1.3 "While" loops

Often times you don't want to repeat yourself. Not only is repeating code ugly and hard to read, but sometimes you don't know how many times you'll need to repeat a certain block of code. Not only that, but if you copy and paste code multiple times, you risk introducing **copy-paste errors**, where you don't spot the bug in a piece of code when using it in a new context. This idea is so important in software development that there is an acronym, **DRY**, for **Don't Repeat Yourself**, which is one of the most upheld principles of writing good code.

Loops are one of the easiest ways you can avoid repeating yourself. A loop is exactly that - a piece of code over which the program loops, repeating it until a certain condition is satisfied (or occasionally, indefinitely).

While loops are the simpler of the two standard loops in Python. For as long as a certain statement continues to be true, they will run the designated output. Here's an example, where we specify a number, and perform a piece of code printing the square[2] of that number until it is sufficiently large.

```
# Initialise the variable.
n = 1

# Start the loop.
while n <= 10:

    # Print "n squared is .."
    print(str(n) + " squared is " + str(n**2) + ".")

    # Increase n by one.
    n = n + 1

>> 1 squared is 1.
>> 2 squared is 4.
>> 3 squared is 9.
>> 4 squared is 16.
>> 5 squared is 25.
>> 6 squared is 36.
>> 7 squared is 49.
>> 8 squared is 64.
>> 9 squared is 81.
>> 10 squared is 100.
```

It's important when using a **while** loop that you don't accidentally create a runaway program. If you don't remember to increase n by one each time here, then your program will run as fast as it can, indefinitely repeating "1 squared is 1", which, if nothing else, just makes you feel a bit sorry for your computer.

By the way, you can also write `n += 1` as a shorthand for `n = n + 1`.

While loops are fast, but it's definitely possible to create accidentally infinite while loops. If this happens, your whole program will never stop, and it will eat up lots of your computer's memory. Because of this, often it's easier (and safer) to use the other main loop provided in Python.

3.1.4 "For" loops

The **for** loop is a very useful tool in Python, as it helps us avoid having to worry about *how many times* we need to loop through a block of code. Often when we repeat some code, we want to do it *for* every element in a list, or *for* each letter in a string, or *for* every x in some y. In this case, so long as the y is a kind of object called an **iterable**, then we can "iterate over" y and repeat the code for each thing in y.

This sounds a bit abstract, so let's give an example. Suppose we have a list of numbers, and *for* each number in the list, we'd like to print the square of that number

```
# Initialise the list of numbers.
numbers_to_square = [1, 2, 7, 3, 6, 9]

# Loop over these numbers.
for number in numbers_to_square:
    print( str(number) + " squared is " + str(number ** 2) + "!")
```

[2]Note that $**$ means exponentiation in Python - using the caret □ doesn't give the right behaviour here!

```
>> 1 squared is 1.
>> 2 squared is 4.
>> 7 squared is 49.
>> 3 squared is 9.
>> 6 squared is 36.
>> 9 squared is 81.
```

When running a *for* loop, we pass an iterable object which we'd like to loop over and give a name to the variable which will be changing as we pass through the list. Then comes the block of code to be executed using this variable.

The syntax is this:

```
for ITEM in ITERABLE:
    <Do something with ITEM>
```

Here's another example.

```
# Create a set of things.
my_favourite_things = {"cats", "dogs", "cupcakes"}

# Loop over these things
for favourite_thing in my_favourite_things:
print("I really like " + favourite_thing + ".")

>> I really like dogs.
>> I really like cats.
>> I really like cupcakes.
```

You'll find *for* loops to be an unquestionably powerful tool for working with long lists of data.

3.2 Modules, Libraries and Packages

3.2.1 Modules

Often when programming you'll want to re-use code that you wrote at an earlier point. For example, you might write a function in one script and then find that you want to re-use that function in a different script. Naturally it doesn't make a lot of sense to copy and paste the definition of that function from the old script to the new script, so, in general, we make use of `import` to make use of code from another script.

If you have another file in your directory which is a python script ending in .py, you can run all of the code in that script while running code in the main script. Suppose we run a piece of code called `my_script.py` which imports a file called `my_functions.py`. Then all of the code in `my_functions.py` will be executed at the point of import, making all of the functions and objects defined in `my_functions.py` available when running `my_script.py`. How do we do this? In `my_script.py` you can simply write:

```
import my_functions
```

Note that you do not need to include the `.py` at the end of the import. Suppose that `my_functions.py` contains the following code:

```
def print_name(name):
    print("My name is " + name + "!")
```

In the same folder we run our script `my_script.py` which contains the lines:

```
# Import the code from my_functions
import my_functions

# Print out my name
my_functions.print_name("Keenan")

>> My name is Keenan!
```

Note that the code that we imported moved the name of the function to "my_functions.print_name". This is the usual behaviour when using `import`, because otherwise we would run the risk of importing a function or object which has the same name as something already in our code. If you want to circumvent this and import the function with the exact same name, you can use:

```
from my_functions import print_name
```

This will make the function `print_name` available as-is in your code, without having to worry about remembering which script it came from. A single file from which we import code is called a **module**. In this case, we imported the **my_functions** module so that we could access the function that was stored inside.

When you're working with code in a large project, you're likely to have a large collection of modules which you might need to access. The entire body of code in your project is called a **library**, consisting of many different modules of code.

3.2.2 Packages

Often it's incredibly useful to access code that someone else has already written. There is an enormous number of open-source projects which we can access, giving us access to lots of incredibly useful code that will speed up your workflows enormously. To see this, we're going to access two important packages called **pandas** and **NumPy**.

You might remember that we already used a tool called **pip** to install **Jupyter Notebook**. When we did this, all we had to do was go to the command line and type `pip install notebook`. When we did this, we were actually looking up the project named `notebook` from a remote collection of projects called the **Python Package Index, PyPI** [2]. You can find out more about it at `https://pypi.org/`.

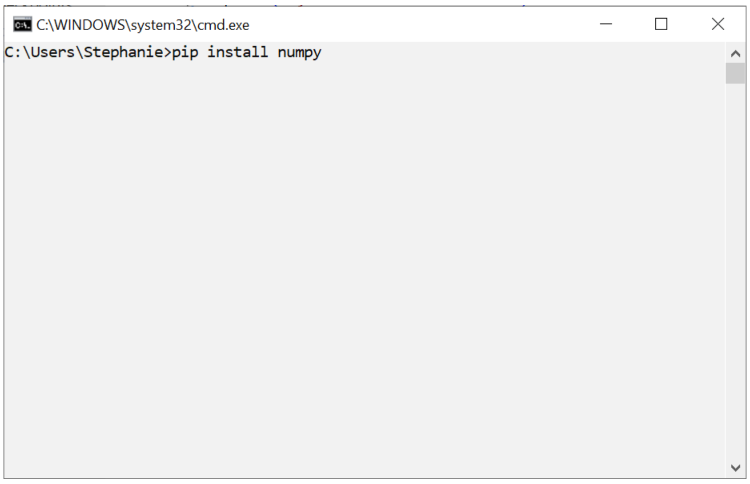
Navigate back to your command-line by re-opening **Command Prompt** or **Terminal**. From there, you can install the pandas and numpy packages by running these two commands, and responding yes (y) when prompted.

```
>> pip install numpy
>> pip install pandas
```

You don't need to navigate to a particular directory - the packages will be installed for you in the correct location (which will be either for all users or local to you, but either way you should be able to access it). In actuality, the pandas package is built on numpy (i.e. pandas has Numpy as a **dependency**), so installing pandas should also automatically install numpy, but we're being explicit with it here for now.

A quick note: when you get further into your programming careers, you might find that you need to install lots of different packages for different projects. Having a lot of packages installed can sometimes cause problems, as they might interact with each other in ways that you were not expecting. To get around this, you might consider managing your packages into different **environments**. An **environment** is a collection of different packages which you can *load* and *unload*, so that those loaded packages act as if they are installed, and the unloaded ones will not. One of the best package and environment managers is **Anaconda**, which you can install online at https://www.anaconda.com/download. We won't go into managing environments much here, but know that in general it's good practice to use them if you're working on large projects or want to become a better developer.

Next time we'll be getting hands-on with using numpy and pandas to explore around with some data. See you all then!



```
C:\Users\Stephanie>pip install numpy
```

4 Distributions and Data

When working with data, we often find that we want to answer questions. In an optimal world, we should operate by taking our assumptions, formulating a statement about what we think might be true about the data, and then we see whether or not the data supports the truth of our statement. Over the next couple of lectures we're going to learn how we can use data to answer these kinds of questions.

Working in this way is the mathematically optimal way to do so, but it's not always the most convenient. Indeed, we don't go through our daily lives formulating hypotheses about every fact in our waking world, and then searching for data to confirm or negate them. However, when working in a scientific, academic or business capacity, we have to make sure that the patterns we think we see actually *exist*, as humans are especially susceptible to seeing patterns which are not, in reality, present.

There are many reasons this can happen. If you look hard enough at a collection of data for long enough, it's always possible to construct some kind of narrative trend in the data. For that reason, rather than looking at the data and finding a narrative trend, you should locate a narrative trend *first*, and then find some *new* data to confirm it.

The narrative you construct hinges implicitly on what **information** you think have about the world. When you write down the information you think you have, what you'll find is that it should describe a **distribution** that you should see in the new data.

4.1 Distributions

A **distribution** in statistics is a mathematical model which represents the expected frequency with which you observe certain events to occur. For example, if I take a random sample of the world's men, I'll notice that the vast majority of them are close to average male height, and far fewer people are very tall or very short.

There are two broad classes of distribution - **discrete** distributions and **continuous** distributions. A *discrete* distribution usually reflects the probability of a finite number of possibilities - if I roll a die, say, then there are exactly 6 outcomes which we can observe. Each of these probabilities can be thought of as representing the expected amount of times we'd expect to see that outcome if we rolled the dice repeatedly. We can then specify the distribution by specifying the probability of each outcome x of the variable X:

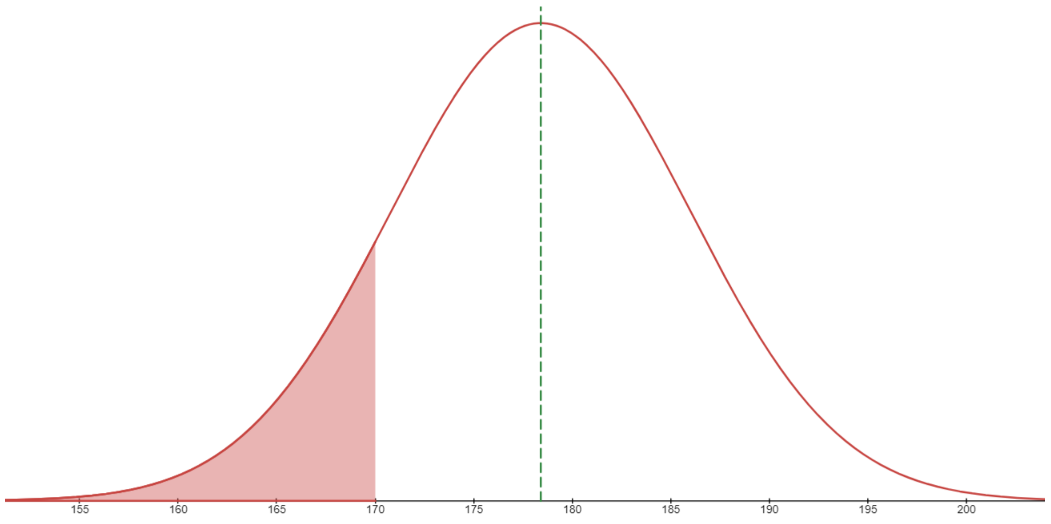
$$p(X = x) = 1/6$$

There are a few different discrete distributions that often come in handy, but the **uniform** distribution is one of the most frequently appearing.

A **continuous** distribution varies smoothly in possible outcomes. For example, if we measure the heights of different people, the possible numbers we could get as output can vary smoothly. If you've done much statistics in school already, you'll likely have met the **normal** distribution, which is the most commonly occurring distribution in nature. We'll briefly discuss this distribution again here, and then we're going to try and look at, model and experiment with data. To do this, we'll use three packages: NumPy, Pandas and Matplotlib.

4.1.1 The Normal Distribution

The normal distribution, also called the Gaussian distribution, is one of the most frequently occurring distributions in nature. Suppose that we, as above, take a large sample of people and look at their heights. We're going to plot the **probability density function** of the normal distribution for men's heights. This is essentially a curve which describes how common it is for men's heights to be around a certain value. The total area under the curve is equal to 1, reflecting all men, or 100%. The **probability** that any given man's height is less than 170cm is given by the area of this shaded region:



The average height is given by 178.4cm, and another number, the standard deviation σ, is 7.6cm. This number represents how spread out the data is; the larger the standard deviation σ, the more variance in height there is among men. If it were very small, then nearly every man would have a height very close to 178.4cm.

While you don't need to remember it, the equation of the curve for this distribution (the probability density function is given by

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

where σ is the standard deviation and μ (mu) is the mean. Data can have lots of different kinds of distributions, but this one is the most common one you're likely to see in nature. We'll stick with it mostly for now, but we'll take a look at how we can think about more complex distributions later on.

4.1.2 A brief recap

Just in case you haven't met the standard deviation yet, it is a measure of how **spread out** the data is, and when we're looking at the entirety of a population, it is given by

$$\sigma = \sqrt{\frac{\sum_{i=1}^{N}(x_i - \mu)^2}{N}}$$

Where N is the size of the population and μ is the population mean. If we want to estimate the standard deviation of a population from a sample, which is more often the case, then this formula gets you closer to the actual value:

$$s = \sqrt{\frac{\sum_{i=1}^{N}(x_i - \mu)^2}{N-1}}$$

The square σ^2 is also known as a variable's **variance**[3].

We would not expect that the variation in the weight of people is at the same scale as the variation in weight of 1kg bags of sugar, and the standard deviation gives us a number with which we can compute this "scale of variation".

4.1.3 Computing a Z-score

In order to work out what area of the diagram is actually shaded, we usually use a helpful tool called a Z-score. The Z-score simply normalises this distribution so that the average becomes zero and the standard deviation becomes one. From here, there are lots of tables that tell you "what the probability is" of observing an individual with a Z-score less than a certain threshold.

For example, converting the height 170cm to a Z-score, we do

$$Z(x) = \frac{x - \mu}{\sigma}$$

and apply this to $170cm$:

$$Z(170) = \frac{170 - 178.4}{7.6} = -1.11$$

Looking this up in a table or online will tell you that the probability of observing a Z-score less than -1.11 is about 13%, so only 13% of men have a height less than 170cm worldwide. As mentioned, there are lots of calculators online to help you with this (you can also use python) - here's one online calculator [6].

4.1.4 Modelling with Numpy and Matplotlib

We're now going to run a virtual experiment and see what happens if we take samples of different sizes of men around the world, and then measure their heights. If you want to produce a simulated sample from a normal distribution, the Numpy package can do this well. We also want to see what we produce, so we're going to use another package, called **matplotlib**, to plot the data in a histogram.

When we import the packages so that we can use their code, it's common to assign them an alias to speed up code production. To do this we write `import numpy as np`, meaning that whenever we want to access the numpy package, we now only need to type `np` in future. We'll also import part of the matplotlib package (pyplot) as `plt`, which is also very standard[4].

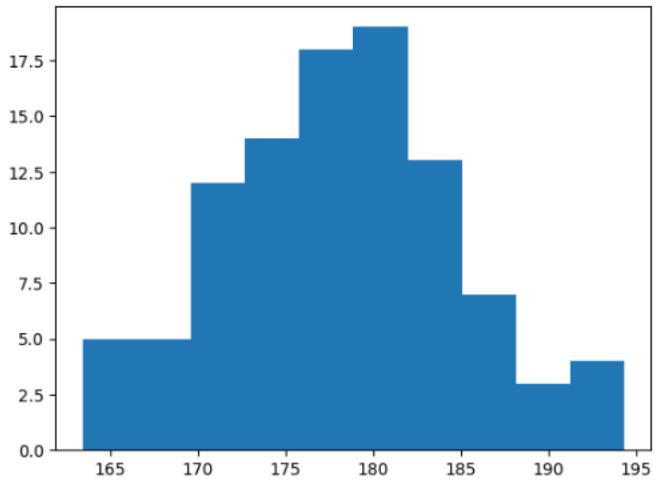
```
# Standard numpy and matplotlib
import numpy as np
import matplotlib.pyplot as plt

# Generate some dummy height data.
mens_heights = np.random.normal(178.4, 7.6, 100)

# Create a histogram of heights.
plt.hist(mens_heights)

# Show the histogram.
plt.show()
```

[3]You might wonder why we use standard deviation rather than an average absolute difference between observations and the mean. There are a few reasons; firstly, the standard deviation is smooth and nice to differentiate. Secondly, variance seems to be the natural choice, as it is additive when we add independent variables together (e.g. Var($X+Y$) = Var(X) + Var(Y)).

[4]Most of these import aliases are pretty standardised - numpy is almost always imported as np, pyplot as plt, and pandas as pd.

Here, the np.random.normal function will produce a certain output called a **Numpy array**. Numpy arrays are a lot like lists, except that they're computationally faster and the types of the entries have to be the same. What we'll get is an array of 100 data points with mean 178.4 and standard deviation 7.6, as if we'd picked 100 men at random worldwide and looked at their heights.

Then `plt.hist` will create a histogram out of the data, and the final command plt.show() will show the data. In a histogram, data is grouped into certain windows, and the **height** of each bar represents how many times the data point we're observing fell inside of that window. Here's the output of running this code.



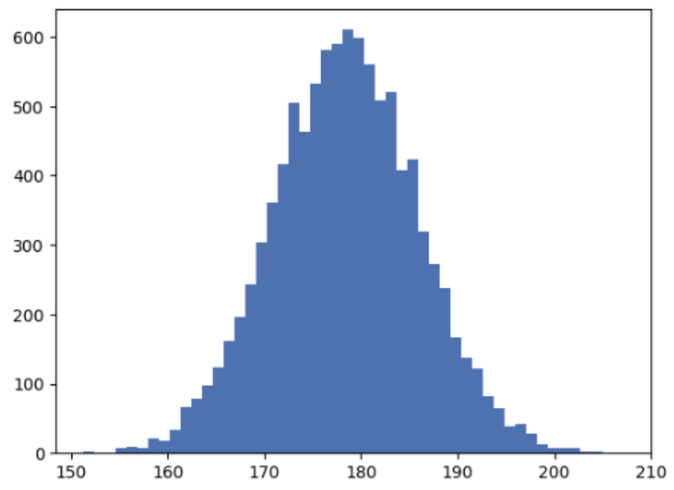
You can see a vague outline of what looks like the normal distribution. In reality, because our sample size was relatively small, there's likely to be a lot of noise in our data - fluctuations in the distribution that are due to chance alone. If we increase now the sample size up to 10000, and break up the histogram into more bins using the keyword argument `bins = 50`, then we get something that looks a lot more like a classical distribution.

```
# Standard numpy and matplotlib
import numpy as np
import matplotlib.pyplot as plt

# Generate some dummy height data.
mens_heights = np.random.normal(178.4, 7.59, 10000)

# Create a histogram of heights.
plt.hist(mens_heights, bins=50)

# Show the histogram.
plt.show()
```

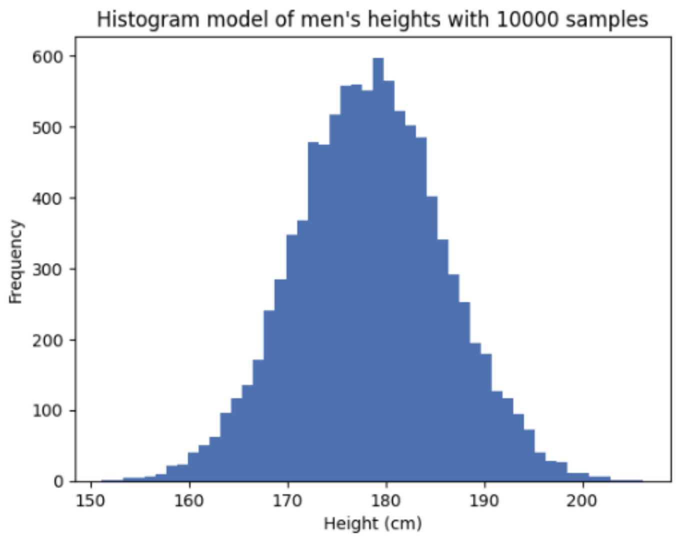
In this code you'll notice that when we called `plt.hist` we also specified the number of bins with `bins = 50`. This is called a **keyword argument** or **kwarg**. While most arguments for python functions are passed according to their order (positional arguments - see for example the documentation for `plt.hist` at https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.hist.html), you can often specify an argument by its name, which you can find in the documentation. Here we specified the **bins** argument should take the value 50 when running the function. In general it's a good idea to have a number of bins roughly equal to the square root of the sample size.

4.1.5 Creating nice plots

Matplotlib comes with a lot of functionality for changing the decoration of your plots to make them more pleasing and easy to read. A good plot should always have a title, scales, and x and y-axis labels to make the graph easier to read. Here are some options we can change.



```
# Change some of the decoration.
plt.xlabel('Height (cm)')
plt.ylabel('Frequency')
plt.title('Histogram model of men's heights with 10000 samples')

# Show the plot again.
plt.show()
```



4.1.6 Modelling new distributions

The data here is completely simulated, but it's often useful to be able to model different kinds of data and distributions. For example, what is the distribution of the **square** of different men's heights? If, instead of looking at the height X, I wanted to try modelling the distribution of weights among men using the fact that I know heights are normally distributed. Suppose then that we have the model that a man's weight is exactly 20 times his height in metres squared. How would the predicted distribution of our model look? We can see very easily, as NumPy arrays make this straightforward.

When we add, subtract, multiply, divide or exponentiate with a NumPy array, the result is another array where that operation is applied to each element, so this makes it quite easy to compute other kinds of distributions.

```
# Standard numpy and matplotlib
import numpy as np
import matplotlib.pyplot as plt

# Generate some dummy height data.
mens_heights = np.random.normal(178.4, 7.59, 100000)

# Convert the data to metres.
mens_heights = mens_heights / 100

# Work out the estimated weight according to our model.
mens_model_weights = 20.0 * (mens_heights ** 2)

# Create a histogram of weights.
plt.hist(mens_model_weights, bins=100)

# Change some of the decoration.
plt.xlabel('Weight according to the model (kg)')
plt.ylabel('Frequency')
plt.title("Histogram model of men's weights with 100,000 samples")

# Show the plot again.
plt.show()

# Print the mean.
print(np.mean(mens_model_weights))

>> 63.80661853622601
```

Try this for yourself and see what you get - play around with making various kinds of distributions! Working with the data in this way can give you an idea of what even really complicated distributions should look like if you know what kinds of assumptions are reasonable to make.

4.2 Working with data

What we have constructed is a model of men's weights, and, quite unusually, we can see that our model predicted the average weight was 63.8 kg, which is almost certainly wrong. How wrong? Well, it's time to maybe have a go at looking at some real data, rather than just simulated data, so that we can compare our model to the actual situation. To do this, we're going to use the other package, `pandas`.

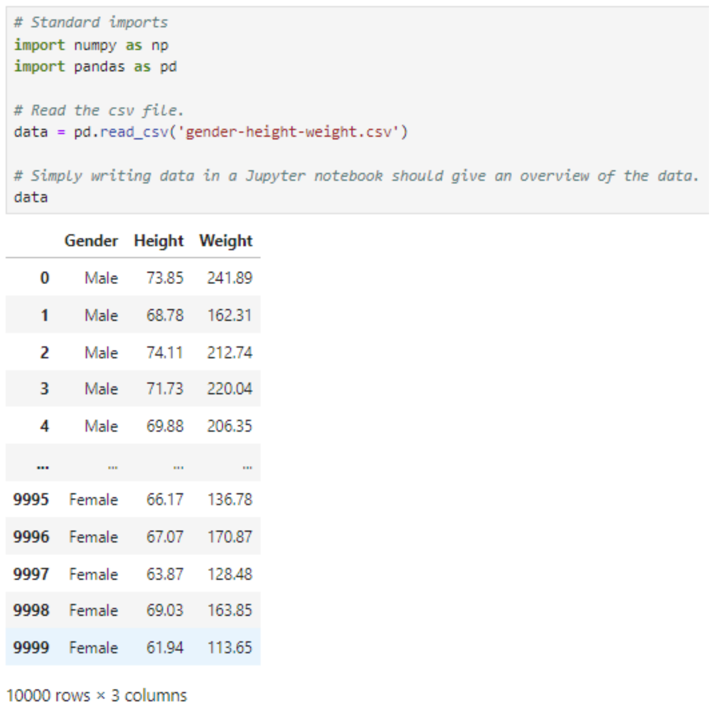
The name pandas is derived from "panel data software" (or so the rumours go), and it allows you to use a really nifty object called the Dataframe, which acts like a large table of data. Further still, it gives you the functionality to import tabular data from software like **excel**.

The dataset we're going to use consists of height and weight data for both men and women, called `gender-height-weight.csv`. I'll try and make this data available on the course pages as well.

If you place the `gender-height-weight.csv` file into the same directory as where you have your Jupyter notebook, then we can access it in the code without having to write out the entire directory of th file. We can then import the data using pandas' `read_csv` function.

You can see here that the pandas `read_csv` function is quite straightforward to call. It will take the data that is contained in `bmi.csv` and save it to a pandas **DataFrame** object, which is a wonderful data structure which represents a python-native version of the DataFrame structure in the **R** programming language. Notice that our data is actually in inches and pounds (as it is an American dataset), so we shall have to convert this back to SI units before we can make a reasonable comparison.

When you import a .csv file or other kind of tabular data, pandas will, by default, assume that the first line contains the column titles. If these are not supplied, you can work around this by providing



```
# Standard imports
import numpy as np
import pandas as pd

# Read the csv file.
data = pd.read_csv('gender-height-weight.csv')

# Simply writing data in a Jupyter notebook should give an overview of the data.
data
```

	Gender	Height	Weight
0	Male	73.85	241.89
1	Male	68.78	162.31
2	Male	74.11	212.74
3	Male	71.73	220.04
4	Male	69.88	206.35
...	...	...	...
9995	Female	66.17	136.78
9996	Female	67.07	170.87
9997	Female	63.87	128.48
9998	Female	69.03	163.85
9999	Female	61.94	113.65

10000 rows × 3 columns

the column names when calling `read_csv`. It will also create a sort of zeroth column, the index, which is sometimes useful in keeping the data structured.

4.2.1 Subsetting the data

We can select particular columns of the data with `data['Height']`, or we can select multiple columns by using a double index: `data[['Height', 'Weight']]` When using pandas, the data in these columns is stored as another datatype called a *pandas series*. For example, you'll see that if we select the `Height` column data with `data['Height']`, the type of what is returned is a *pandas series*.

If we want to get a subset of the data in a pandas DataFrame, we can also put a pandas series consisting of booleans inside of the square brackets, allowing us to filter the data appropriately. To construct such a boolean series, we can apply a logical test to each of the rows in the DataFrame. For example, to get a series which tests whether or not each observation corresponds to a male, we can do the following:

We can then get a new DataFrame of only the men's data using:

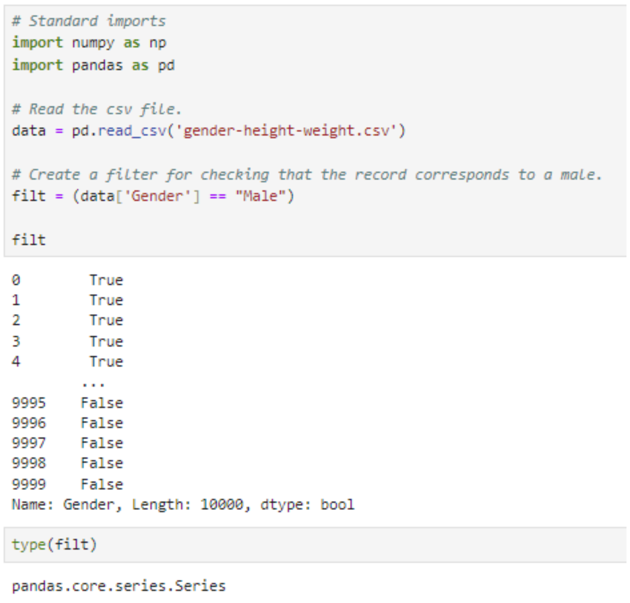
```
# Select the men's heights and weights.
mens_height_weight = data[filt]
```

This will give us a filtered DataFrame `mens_height_weight`, which gives the men's heights and weights in inches and pounds, respectively. Since we want to be able to eventually compare this distribution with our modelled distribution, we're going to need to convert this data to metres and kilograms. To do this, we can simply select the column, apply some operations to convert the data, and save it again to a new DataFrame. It's good practice to create a new DataFrame to save the data to; otherwise you'll get several warnings from pandas that you might cause an error.

Now that we've managed to convert the data, we can try and plot it.

4.2.2 Plotting the data

There are two straightforward ways to plot data using pandas. Firstly, it plays well with Matplotlib, so we can plot it using pyplot. Secondly, pandas also has its own built-in plotting methods. For now, we'll keep using pyplot to keep it simple, but you can read more about the pandas-native plotting

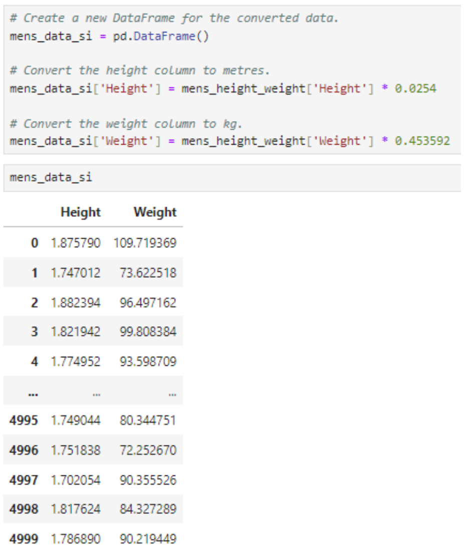


```
# Standard imports
import numpy as np
import pandas as pd

# Read the csv file.
data = pd.read_csv('gender-height-weight.csv')

# Create a filter for checking that the record corresponds to a male.
filt = (data['Gender'] == "Male")

filt
```

```
0        True
1        True
2        True
3        True
4        True
        ...
9995    False
9996    False
9997    False
9998    False
9999    False
Name: Gender, Length: 10000, dtype: bool
```

```
type(filt)
```

```
pandas.core.series.Series
```



```
# Create a new DataFrame for the converted data.
mens_data_si = pd.DataFrame()

# Convert the height column to metres.
mens_data_si['Height'] = mens_height_weight['Height'] * 0.0254

# Convert the weight column to kg.
mens_data_si['Weight'] = mens_height_weight['Weight'] * 0.453592
```

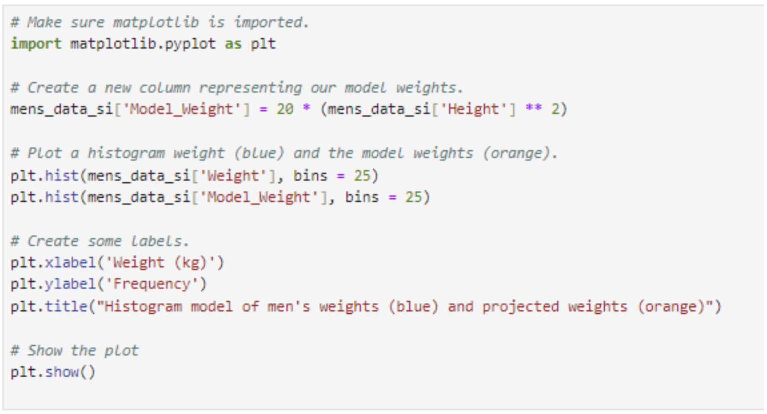
```
mens_data_si
```

	Height	Weight
0	1.875790	109.719369
1	1.747012	73.622518
2	1.882394	96.497162
3	1.821942	99.808384
4	1.774952	93.598709
...	...	...
4995	1.749044	80.344751
4996	1.751838	72.252670
4997	1.702054	90.355526
4998	1.817624	84.327289
4999	1.786890	90.219449

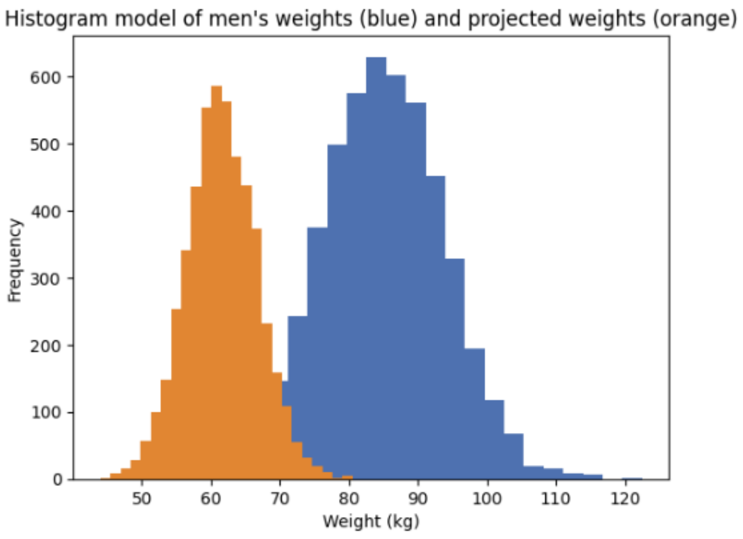
here: https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.plot.html, at the documentation page for the plot method.

Since we've converted the data to the same units, we can now compare the weights that our model predicted according to height with the actual weights that we see in the data. By default, using `plt.hist` twice will plot two different histograms on the same axes, allowing us to compare the distributions. Let's see what we get.

As you can see, our model was definitely not a good model whatsoever! Our model essentially assumed that everybody has a BMI of exactly 20, which is definitely not the case. We'll have a look next time at how we can use python to perform statistical tests, to show conclusively (and scientifically) that our model is wrong!



```
# Make sure matplotlib is imported.
import matplotlib.pyplot as plt

# Create a new column representing our model weights.
mens_data_si['Model_Weight'] = 20 * (mens_data_si['Height'] ** 2)

# Plot a histogram weight (blue) and the model weights (orange).
plt.hist(mens_data_si['Weight'], bins = 25)
plt.hist(mens_data_si['Model_Weight'], bins = 25)

# Create some labels.
plt.xlabel('Weight (kg)')
plt.ylabel('Frequency')
plt.title("Histogram model of men's weights (blue) and projected weights (orange)")

# Show the plot
plt.show()
```



5 Hypothesis Testing

We mentioned last time that we'd like to have a more scientifically rigorous way of justifying that our model is wrong. When scientists make hypotheses, they have to be careful to make sure that they don't fit their data to match a pattern, or search for spurious patterns in data. To do this, the best way to work is to construct a hypothesis and then apply a statistical test on some data to see if it matches with your hypothesis. This way we avoid the difficult and murky waters of searching for patterns to report, which can often lead you to imaginary results.

5.1 Null Hypotheses and Alternative Hypotheses

When we formulate a hypothesis, we set up two competing hypotheses and see which one the data prefers. These two hypotheses are called the **null hypothesis** (often denoted H_0) and the **alternative hypothesis** (often denoted H_1). They are mutually exclusive, so whenever the null hypothesis is true, the alternative hypothesis must be false and vice versa.

The **null hypothesis** is a statement which reflects the status quo or a baseline assumption. For example, if you want to test that a new medication for Alzheimer's has an effect on cognitive scores, your null hypothesis would be that it has no effect. The **alternative hypothesis** is the negation of the null hypothesis and states what must be true if the null hypothesis is false.

H_0 The mean cognitive score of Alzheimer's patients subjected to the treatment is the same as the mean cognitive score of Alzheimer's patients who were not subjected to the treatment.

H_1 There is a difference in the mean cognitive score of Alzheimer's patients subjected to the treatment and the mean cognitive score of Alzheimer's patients not subjected to the treatment.

When we perform a *hypothesis test*, we're using the data we have to see if there is enough evidence to accept H_1 over H_0 (that is, to reasonably believe that H_1 is true and H_0 is false), within a certain margin of error. This is the basis for almost all hypothesis testing - there is usually a default, status-quo assumption H_0, and a mutually exclusive alternative H_1, which is often more interesting.

5.2 p-values

We mentioned in the previous subsection that you might incorrectly conclude that H_1 is true and H_0 is false because of a *margin of error*. For example, if we're quite unlucky, it could be possible that the data we collected was simply noisy, and by chance we saw evidence in favour of H_1 (the break from the status quo) when in fact this deviation might have just been due to noise, randomness or chance. We represent this *margin of error*, the chance that we wrongly accept H_1 over H_0, with something called a **p-value** - which is a probability.

To correctly perform a hypothesis test, we have to specify a certain p-value before the test. We set up the test so that there is a certain probability p that we incorrectly accept H_1 over H_0. If we have a very small value of p and the test turns out to rule in favour of H_1, this means that the likelihood that the hypothesis H_1 is false is very small. *However*, extraordinary claims require extraordinary evidence, and it's much more likely that with a small p-value, your test will not conclusively find the **significant difference** you're hoping to see.

For this reason, the selected p-value for the test is usually relatively standardised within a field of knowledge. In most scientific disciplines, a p-value of 0.05 is usually taken as the standard value. That is, hypotheses are normally tested so that there is only a 5% chance of incorrectly rejecting H_0 in favour of H_1. When there's a lot on the line, however, for example a need to restructure the fundamental language of physics (we're looking at you, *Higgs Boson*), then a much smaller p-value might be required. In validating the existence of the Higgs Boson, researchers used a 5σ ($p = 0.0000003!$) significance test. Even then, it's still *theoretically* possible that the observed data is due to chance [5].

5.3 Smaller values of p

It's been discussed in the scientific literature whether or not the industry standard of 5% should be swapped for a smaller number like 1% [14]. Many academic papers will publish results containing multiple tests, and it won't surprise you that if you do 20 tests at the 5% significance level, you can expect at least one of them to be significant (even if there is no relationship to be found at all). Going even further, the main result of maybe 1 in every 20 papers might be just due to noise. For this reason, it's important to be careful when doing statistical tests, and be sure to set the significance value ahead of time. We'll discuss more about how we can manage statistical irregularities shortly. For now, we're going to see how we might do a significance test in python.

5.4 Performing a t-test

We're going to stick with normal distributions for now and look at the most standard hypothesis test you might want to perform, called a **t-test**. In fact, many statistical tests often act as if the underlying distribution is normal, even when it isn't, and you might find yourself using these kinds of tests an enormous amount.

The t-test uses a test statistic T, which we can then look up in a table or online to see the corresponding significance level. The statistic measures the distance between a model, or status-quo mean (e.g. we think the mean weight of cheeses in a factory is 1kg), and the mean of the actual distribution (e.g. the cheeses actually average a weight of 1.1kg).

To do this, we take the reference mean μ, which reflects our status-quo assumption, and compute the mean $\bar{x}$ of a sample. We'll then compare them using the T-statistic:

$$T = \frac{\mu - \bar{x}}{s/\sqrt{n}}$$

where s is the sample standard deviation, and n is the total number of observations in the sample.

5.4.1 A cheesy example

Let's go with the cheese example. Suppose that **Carmine's** manufactures large wheels of cheese which, on the label, are supposed to be $1kg$. The CEO, Avril, wants to see whether or not the cheese that they're manufacturing is actually this size. To do this, Avril takes a sample of 10 wheels of cheese and records their weights in a spreadsheet, `cheese_10.csv`. For each sample, they record the weight in kilograms, the calcium content in grams, and the vitamin D content in micrograms (mcg).

To see if the cheese, does, in fact, have a mean weight of 1kg, they perform a t-test to see if the mean value 1 fits with the data. While it's reasonable to compute the test statistic T by hand with 10 samples, it might not scale up well with more samples, so we can use python to do this for us.

The function we're going to use is part of the `scipy` package, so you might want to install this first before proceeding if you want to follow along. Scipy is useful for lots of scientific work in python, especially for statistical analysis.

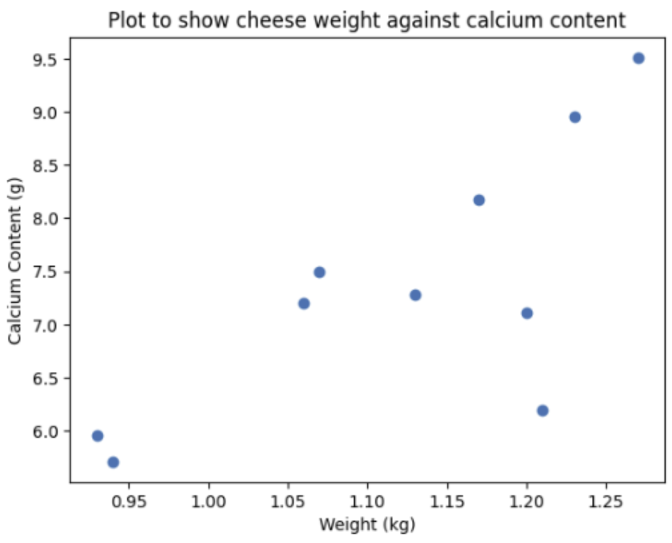
```
# Import numpy and pandas.
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

# Import a special function for a 1-sample t-test in scipy.
from scipy.stats import ttest_1samp

# Import the data.
cheese_data = pd.read_csv('cheese_10.csv')

# Plot the data
plt.scatter(cheese_data['Weight_kg'], cheese_data['Calcium_g'])
plt.xlabel('Weight (kg)')
plt.ylabel('Calcium Content (g)')
plt.title('Plot to show cheese weight against calcium content')

# Show the data.
plt.show()
```

As you can see, this plot isn't particularly informative, but it might give us some indication that calcium content improves with the weight of the cheese slightly. We'll now try and use the `ttest_1samp` function to see if, at the 5% significance level ($p = 0.05$) we can declare that the weight is, in fact, not equal to 1kg in general. The hypotheses are set as follows:

H_0 The mean weight of a manufactured wheel of cheese is 1 kg.

H_1 The mean weight of a manufactured wheel of cheese is not 1 kg.

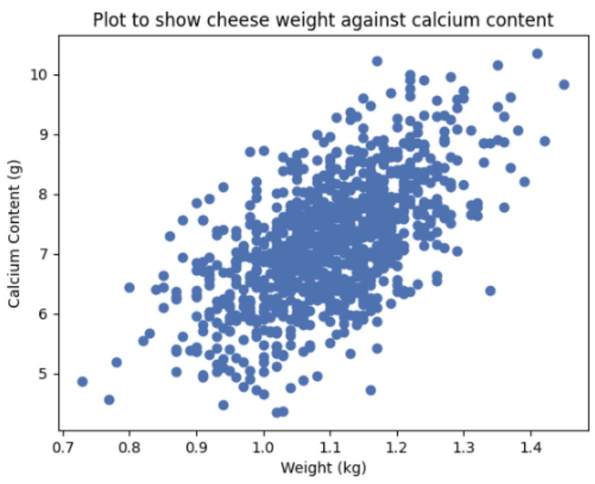
The function takes a sample as the first argument, and then the status-quo population mean as the second argument. It can return both the T-statistic and the p-value at the same time. To assign these both simultaneously we can write a comma on the left-hand side of the assignment operator (=):

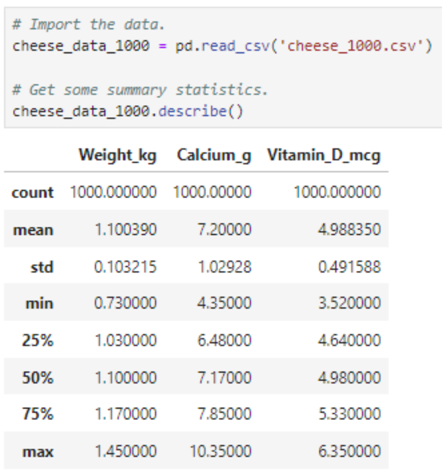


```
# Get the t_statistic and the p_value.
t_statistic, p_value = ttest_1samp(cheese_data['Weight_kg'], 1.0)

# Round the t-statistic and p-value.
t_statistic = np.round(t_statistic, 4)
p_value = np.round(p_value, 4)

# Print these out.
print("The t-statistic is " + str(t_statistic) + ".")
print("The p-value is " + str(p_value) + ".")

>> The t-statistic is 3.2301.
>> The p-value is 0.0103.
```

Since the corresponding p-value we calculated is less than 0.05, we reject the null hypothesis, and we accept the fact that the mean weight of the cheese is not actually equal to 1 kg. But what should that weight be? To get this information, Avril takes a new sample of 1000 wheels of **Carmine's** cheese. They store the results in `cheese_1000.csv`.

We'll load the data again, and use the `pd.DataFrame.describe()` method to get information about the distributions of the sample.





```
# Import the data.
cheese_data_1000 = pd.read_csv('cheese_1000.csv')

# Get some summary statistics.
cheese_data_1000.describe()
```

	Weight_kg	Calcium_g	Vitamin_D_mcg
count	1000.000000	1000.00000	1000.000000
mean	1.100390	7.20000	4.988350
std	0.103215	1.02928	0.491588
min	0.730000	4.35000	3.520000
25%	1.030000	6.48000	4.640000
50%	1.100000	7.17000	4.980000
75%	1.170000	7.85000	5.330000
max	1.450000	10.35000	6.350000

Reading off the sample mean from the `.describe()` method, we can see that the mean weight of a wheel of cheese is, in fact, 1.1kg. Better update the labels! But in addition to the mean and standard deviation (std), we're given the minimum and maximum values, and three quartiles. I.e. 25% of the cheese blocks have a weight less than 1.03kg. The lowest one is ridiculously small - 0.73kg. Avril needs to make their cheese production more standardised!

5.5 Other statistical tests

Depending on what you want to test, there are lots of different kinds of statistical tests. We're going to give a brief outline of various different statistical tests here, but we won't go much into detail.

5.5.1 Variations on the t-test

There are various different scenarios where we might want to compare means between normal distributions, or distributions that we expect are normally distributed. The example we gave above was just one version; in particular, we checked the test statistic (the mean) of a sample against a reference value. This is called a **one-sample t-test**.

What if we want to compare the means of two independent samples (i.e. they're from different populations, e.g. Alzheimer's patients versus healthy controls), and see if they are the same? In such a case we use a **two sample t-test**. The corresponding function in `scipy` is `scipy.stats.ttest_ind`. The two-sample t test assumes that both variables are distributed normally and *with equal variance*, so the standard deviation σ is the same.

If we want to look at statistical differences at the individual level (e.g. we perform a study where we observe the same person twice, before and after an intervention, say), then we use a **paired t-test**. In this scenario we expect that the differences between individuals before and after are normally distributed. We do not strictly require that the two distributions have the same variances.

5.5.2 Tests with more than two samples

One of the most standard tests for checking to see if one of several sample means is different to the others is an ANOVA (short for Analysis of Variance). In this scenario, we have three or more independent groups, and our null hypothesis is that all of their population means are the same.

There are a few variations. A **one-way ANOVA** measures one kind of dependent variable (e.g. cognitive scores) across groups in a categorical variable (e.g. Alzheimer's vs Mild Cognitive Impairment vs Control). You can do this in python with `scipy.stats.f\_oneway`.

A **two-way ANOVA** uses two different kinds of categories. For example, maybe we take age bracket and income bracket against a test score. In this scenario we actually have three null hypotheses. Firstly, that the group means across age brackets are equal; secondly, that the group means across income brackets are equal, and thirdly that the group means across income and age brackets are equal. The first two are **main effect tests** and the last is the **interaction effect test**.

For completeness we'll also mention the **MANOVA**, the **multivariate analysis of variance**. In this scenario there are possibly multiple dependent and independent variables, and we look to see if either of the independent variables changes across main and interaction groups.

5.5.3 Conclusion

There are many different kinds of statistical tests that you can employ in many different scenarios. The bread-and-butter of statistical tests are the t-test and the ANOVA. Shortly we're going to learn about hypothesis testing when we work with data out in the wild - in particular, how we can employ some very powerful modern methods to simulate distributions and use the result to test various hypotheses. Next time, we're going to look at exploratory data analyses - how we can start exploring with data and answering interesting questions.

See you then!

6 Exploratory Data Analysis

We're going to talk a bit now about the exploratory data analysis. When you obtain a new dataset, or want to get to grips with understanding some data from your business or project, then the first thing you'll want to do is get a good feel for the data and start to understand the natural patterns which emerge in your data. This starting process if often referred to as an **exploratory data analysis**, or **EDA**.

There are lots of good reasons to do an EDA. First and foremost, it allows you to spot patterns in your data that you might not have otherwise been aware of. For example, you might see a surprising link between the number of people who bought today's hotdogs in your theme park and customers who went to see the nurse in the afternoon, or a curious correlation between the kinds of medicines your study subjects are taking and how commonly they are affected by various diseases. EDA is a great starting point for making discoveries. Secondly, EDA will let you know pretty quickly if something is amiss in your data. You might notice that the important calculation that was supposed to be done with SI units was done with imperial measurements, and *that's why your climate orbiter might have crashed into Mars* [3].

For that reason, we're going to have a look at the whole process of performing a relatively basic exploratory data analysis, so that you can take what you learn and apply it to your own datasets.

6.1 Locating Data

When you want to work on your own projects, or start putting together a portfolio of small data-science projects to convince a potential employer that you have these kinds of skills, you'll often have to start by sourcing your own data.

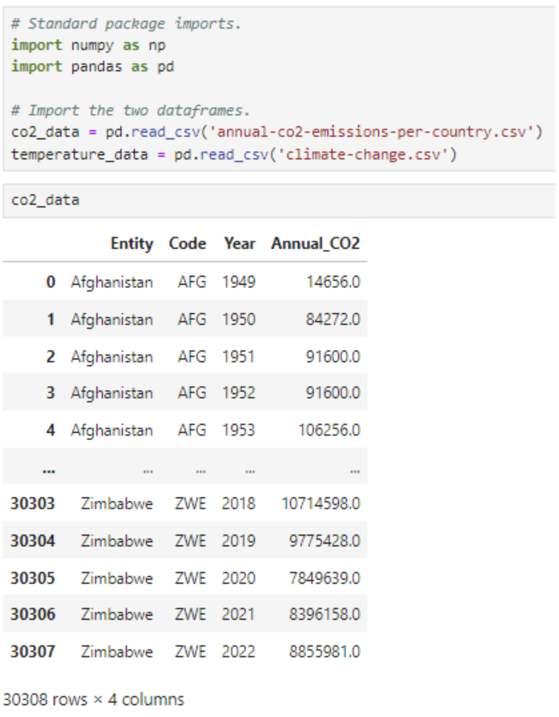
If you're working at a company or other institution, a lot of their data might be stored in a SQL (also sometimes pronounced as "sequel") database. This is a bit beyond the scope of this course, but if you want to find out how to search through SQL databases, you can learn about the query language SQL online. You can even use SQL queries to search through pandas DataFrames with `pd.DataFrame.query`.

We're going to work with some public datasets. There's lots of really interesting data out there, and putting it together gives us a chance to see lots of interesting relationships. Some great places to find data are **Our World in Data**, the **Office for National Statistics** or **Kaggle**.

We're going to look specifically at two datasets from **Our World in Data** looking at the CO_2 production and the global temperature anomaly. We have these two datasets saved as `.csv` files under `annual-co2-emissions-per-country.csv` and `climate-change.csv` respectively.

6.2 Importing Data

We're going to start a new Jupyter Notebook for our exploratory data analysis. We're going to get straight in and use pandas to import the `.csv` data.

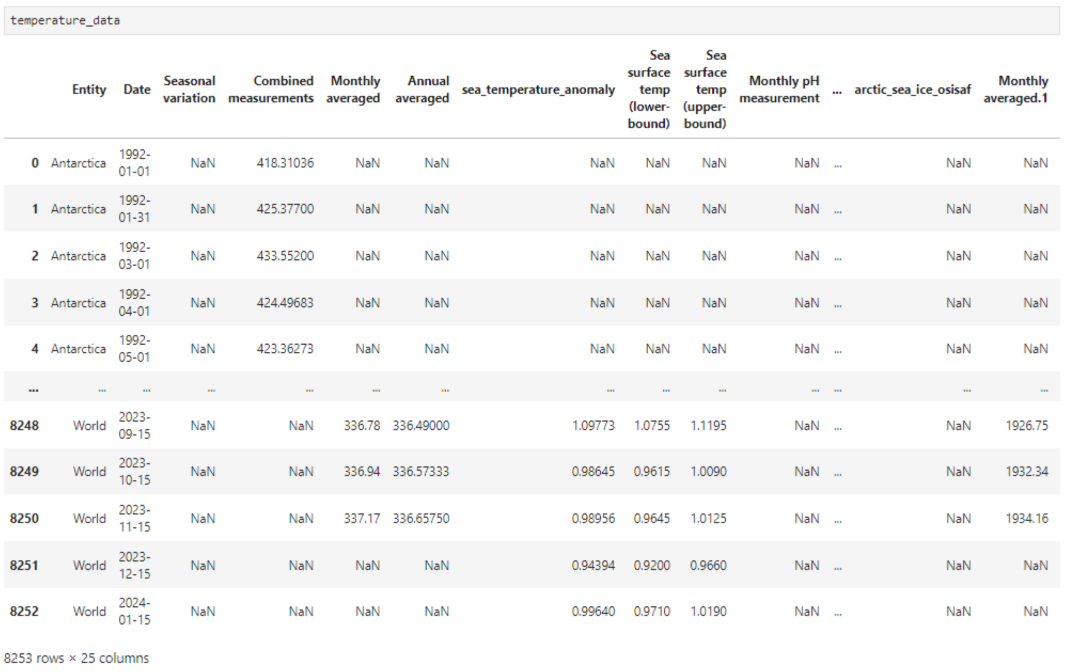


```python
# Standard package imports.
import numpy as np
import pandas as pd

# Import the two dataframes.
co2_data = pd.read_csv('annual-co2-emissions-per-country.csv')
temperature_data = pd.read_csv('climate-change.csv')
```

```python
co2_data
```

	Entity	Code	Year	Annual_CO2
0	Afghanistan	AFG	1949	14656.0
1	Afghanistan	AFG	1950	84272.0
2	Afghanistan	AFG	1951	91600.0
3	Afghanistan	AFG	1952	91600.0
4	Afghanistan	AFG	1953	106256.0
...	...	...	...	...
30303	Zimbabwe	ZWE	2018	10714598.0
30304	Zimbabwe	ZWE	2019	9775428.0
30305	Zimbabwe	ZWE	2020	7849639.0
30306	Zimbabwe	ZWE	2021	8396158.0
30307	Zimbabwe	ZWE	2022	8855981.0

30308 rows × 4 columns



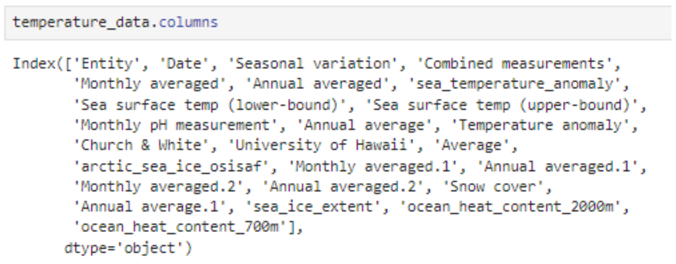
```python
temperature_data
```

	Entity	Date	Seasonal variation	Combined measurements	Monthly averaged	Annual averaged	sea_temperature_anomaly	Sea surface temp (lower-bound)	Sea surface temp (upper-bound)	Monthly pH measurement	...	arctic_sea_ice_osisaf	Monthly averaged.1
0	Antarctica	1992-01-01	NaN	418.31036	NaN	NaN	NaN	NaN	NaN	NaN	...	NaN	NaN
1	Antarctica	1992-01-31	NaN	425.37700	NaN	NaN	NaN	NaN	NaN	NaN	...	NaN	NaN
2	Antarctica	1992-03-01	NaN	433.55200	NaN	NaN	NaN	NaN	NaN	NaN	...	NaN	NaN
3	Antarctica	1992-04-01	NaN	424.49683	NaN	NaN	NaN	NaN	NaN	NaN	...	NaN	NaN
4	Antarctica	1992-05-01	NaN	423.36273	NaN	NaN	NaN	NaN	NaN	NaN	...	NaN	NaN
...	...	...	...	...	...	...	...	...	...	...	...	...	...
8248	World	2023-09-15	NaN	NaN	336.78	336.49000	1.09773	1.0755	1.1195	NaN	...	NaN	1926.75
8249	World	2023-10-15	NaN	NaN	336.94	336.57333	0.98645	0.9615	1.0090	NaN	...	NaN	1932.34
8250	World	2023-11-15	NaN	NaN	337.17	336.65750	0.98956	0.9645	1.0125	NaN	...	NaN	1934.16
8251	World	2023-12-15	NaN	NaN	NaN	NaN	0.94394	0.9200	0.9660	NaN	...	NaN	NaN
8252	World	2024-01-15	NaN	NaN	NaN	NaN	0.99640	0.9710	1.0190	NaN	...	NaN	NaN

8253 rows × 25 columns

This is a good time to mention that pandas can also handle some other kinds of imports. For example, it also has the `read_excel` method for reading `.xls` or `.xlsx` files.

After you import the data, it's a good idea to get a little bit of awareness of what is contained in the data. Have a look at the column titles and the number of entries. You'll see from the figures that the `co2_data` DataFrame has considerably fewer columns, whereas the `temperature_data` has 25 columns! To get a full list we can use `temperature_data.columns` to get a complete list.

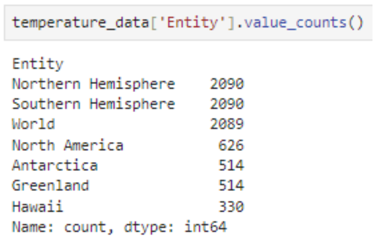


```
temperature_data.columns
```

```
Index(['Entity', 'Date', 'Seasonal variation', 'Combined measurements',
       'Monthly averaged', 'Annual averaged', 'sea_temperature_anomaly',
       'Sea surface temp (lower-bound)', 'Sea surface temp (upper-bound)',
       'Monthly pH measurement', 'Annual average', 'Temperature anomaly',
       'Church & White', 'University of Hawaii', 'Average',
       'arctic_sea_ice_osisaf', 'Monthly averaged.1', 'Annual averaged.1',
       'Monthly averaged.2', 'Annual averaged.2', 'Snow cover',
       'Annual average.1', 'sea_ice_extent', 'ocean_heat_content_2000m',
       'ocean_heat_content_700m'],
      dtype='object')
```

It's often helpful when you're looking at the data to use Jupyter's ability to render Markdown. A great way to do this is to break your work up into sections and write labels for what your code is broadly doing or trying to achieve. You might even find it helpful to label your sections with questions, to help guide your exploration.

Looking at what we find in `temperature_data.columns`, we realise we have a column called **Entity**, just like in the `co2_data` DataFrame. A useful command for understanding what different kinds of entries appear in a column is `value_counts()`. Here's what we get:



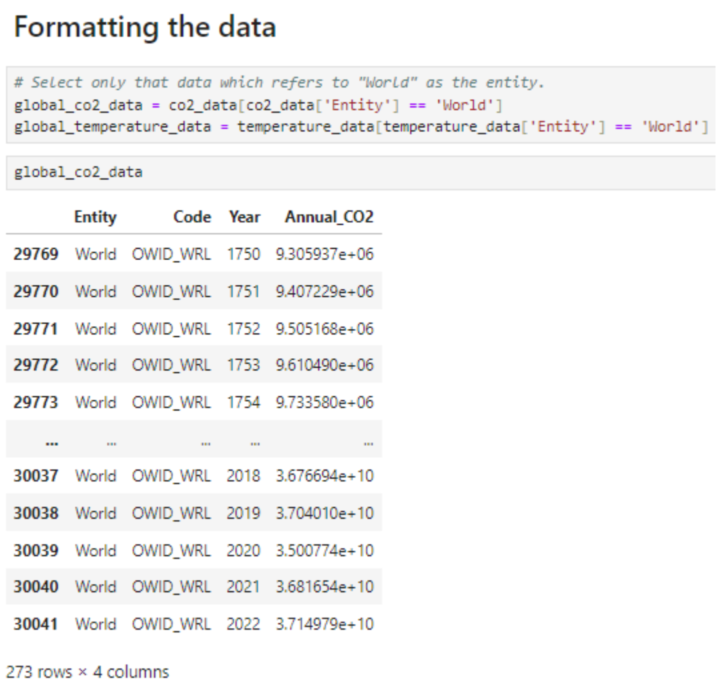
```
temperature_data['Entity'].value_counts()
```

```
Entity
Northern Hemisphere    2090
Southern Hemisphere    2090
World                  2089
North America           626
Antarctica              514
Greenland               514
Hawaii                  330
Name: count, dtype: int64
```

With this in hand, we can now see that there are entries for different parts of the planet, with different records for different regions. We're going to be focusing on global emissions and temperature data. Indeed, checking which values of **Entity** are available in the `co2_data` with `co2_data['Entity'].unique()` shows us that there are also CO_2 measures for the entire world. Let's move on to getting the data into a format that we can work better with.

6.3 Cleaning and formatting the data

Since we're only going to worry about collecting the global data for now, we're going to subset the data which is explicitly referring to "World" as the entity in both datasets. We'll save the resulting data to `global_co2_data` and `global_temperature_data` respectively.



Formatting the data

```
# Select only that data which refers to "World" as the entity.
global_co2_data = co2_data[co2_data['Entity'] == 'World']
global_temperature_data = temperature_data[temperature_data['Entity'] == 'World']
```

```
global_co2_data
```

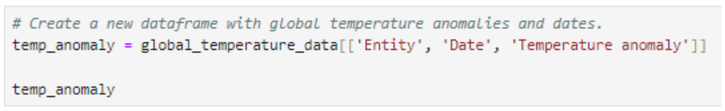
	Entity	Code	Year	Annual_CO2
29769	World	OWID_WRL	1750	9.305937e+06
29770	World	OWID_WRL	1751	9.407229e+06
29771	World	OWID_WRL	1752	9.505168e+06
29772	World	OWID_WRL	1753	9.610490e+06
29773	World	OWID_WRL	1754	9.733580e+06
...	...	...	...	...
30037	World	OWID_WRL	2018	3.676694e+10
30038	World	OWID_WRL	2019	3.704010e+10
30039	World	OWID_WRL	2020	3.500774e+10
30040	World	OWID_WRL	2021	3.681654e+10
30041	World	OWID_WRL	2022	3.714979e+10

273 rows × 4 columns

In this part of the data, all of the times are recorded in **years**. We'd like to see what the relationship is between yearly CO_2 production and the movement of temperature anomaly, so we need to make sure that the times match up correctly. We can see that for the temperature data the date is recorded with a month and a day. To get around this, we're going to average across the year to get an average value for the temperature anomaly. To do that we're going to use a **pivot table**.

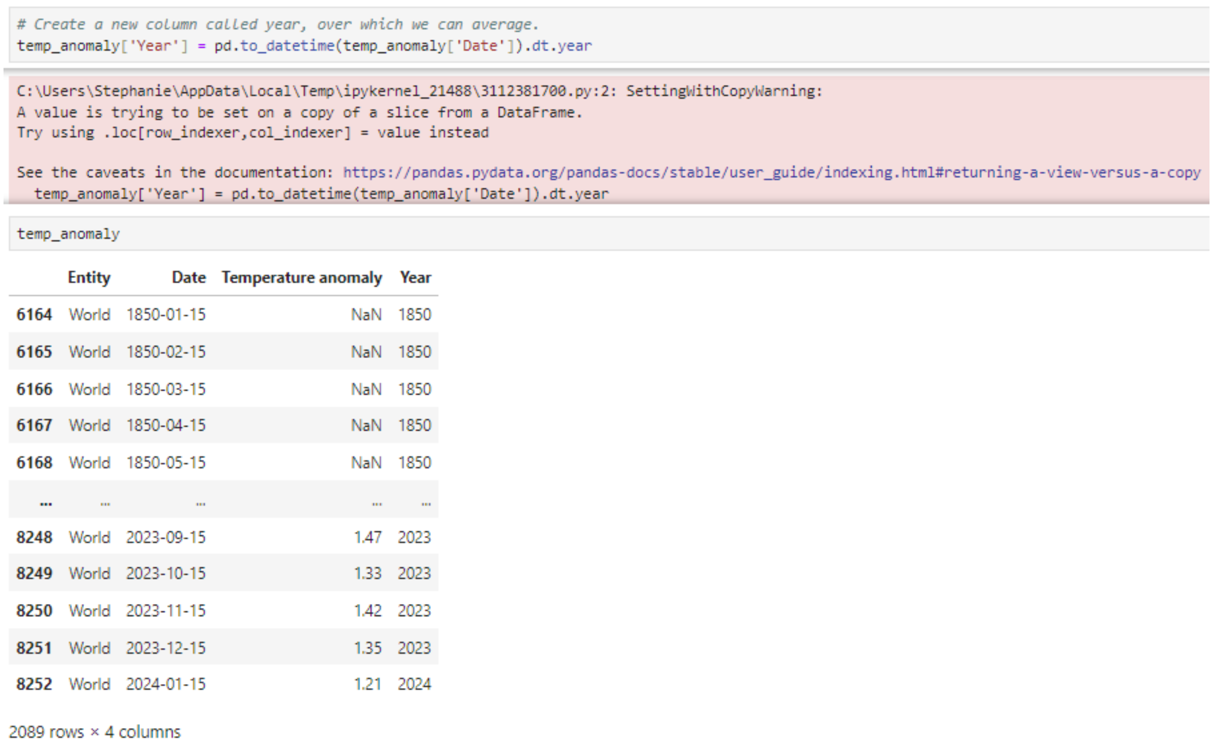
A **pivot table** is a representation of data which allows you to compare groups based on different characteristics. You specify certain **values** - numerical data that you want to understand between groups, and the **groups** into which you want to break the data. By default, `pd.pivot_table` will average the numerical values across the group, which is what we want to do here.

To start, we're going to filter out all of the columns we don't need from the global temperature data DataFrame down to another DataFrame called `temp_anomaly`.



```
# Create a new dataframe with global temperature anomalies and dates.
temp_anomaly = global_temperature_data[['Entity', 'Date', 'Temperature anomaly']]

temp_anomaly
```

Next, we will add a new column onto the DataFrame called **Year**. To achieve this, we're going to change the datatype of the **Date** column to a **datetime**, extract the year using `.dt.year` and saving this to the new column. Don't worry too much about the specifics of this for now. If you want to learn more about time series data (that is, data with a time-point attached), you can find a lot of resources online.



```
# Create a new column called year, over which we can average.
temp_anomaly['Year'] = pd.to_datetime(temp_anomaly['Date']).dt.year
```

```
C:\Users\Stephanie\AppData\Local\Temp\ipykernel_21488\3112381700.py:2: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
  temp_anomaly['Year'] = pd.to_datetime(temp_anomaly['Date']).dt.year
```

```
temp_anomaly
```

	Entity	Date	Temperature anomaly	Year
6164	World	1850-01-15	NaN	1850
6165	World	1850-02-15	NaN	1850
6166	World	1850-03-15	NaN	1850
6167	World	1850-04-15	NaN	1850
6168	World	1850-05-15	NaN	1850
...	...	...	...	...
8248	World	2023-09-15	1.47	2023
8249	World	2023-10-15	1.33	2023
8250	World	2023-11-15	1.42	2023
8251	World	2023-12-15	1.35	2023
8252	World	2024-01-15	1.21	2024

2089 rows × 4 columns

Note that pandas is throwing a warning because it's concerned that we're editing something in place. In this particular case it's being a bit overcautious, as we're actually constructing a new column for the year data. Now we have about 12 entries for each year, and you'll notice that under `Temperature anomaly` we have a number of entries that say `NaN`. This stands for **not a number**, and reflects where data is missing or absent. This is the default way of representing missing data in pandas and numpy, and it makes it easier for different functions to work out what it should be doing with the data.

As we create the pivot table, we're going to see that pandas will automatically discard these entries. Let's go ahead and create the pivot table - the function will return a DataFrame.

```python
# Create the pivot table.
year_anomaly = pd.pivot_table(temp_anomaly, values=['Temperature anomaly'], columns=['Year']).transpose()

# Reset the index.
year_anomaly.reset_index(inplace = True)

# Show the year anomaly dataframe.
year_anomaly
```

	Year	Temperature anomaly
0	1880	-0.154167
1	1881	-0.075833
2	1882	-0.097500
3	1883	-0.162500
4	1884	-0.273333
...	...	...
140	2020	1.011667
141	2021	0.843333
142	2022	0.890833
143	2023	1.165833
144	2024	1.210000

145 rows × 2 columns

The first argument to `pivot_table` is the DataFrame, and we specify the values and groups with `values` and `columns` respectively. Lastly, as the pivot table returns a long horizontal table, we've used `transpose()` to flip the axes, getting it back into a format we were expecting. The only difference is that it is now using the `Year` column as an index. We can reset this using `reset_index(inplace = True)`. The `inplace = True` part simply changes the underlying DataFrame without us having to re-assign it to save the changes.

We're going to make one last addition to the data. Since `global_co2_data` shows the amount of CO_2 produced by different entities in **each year**, and it's more likely that the **total** amount of CO_2 in the atmosphere is affecting the temperature anomaly, we're going to add a **cumulative** column which represents the total amount of carbon dioxide added to the atmosphere since the start of the record. For this we can use `cumsum()` to get a cumulative column.

```python
# Create a new column, the cumulative co2.
global_co2_data['Cumulative_CO2'] = global_co2_data['Annual_CO2'].cumsum()
```

This adds a new column called `Cumulative_CO2` which contains the total amount of equivalent CO_2 that's been added to the atmosphere since the beginning of the record. With this in hand, we're in a great position to actually start exploring the relationship between these two datasets.

6.4 Joining data

We now have two DataFrames containing different data, and to put them back together we're going to learn the data science power move - the **join**.

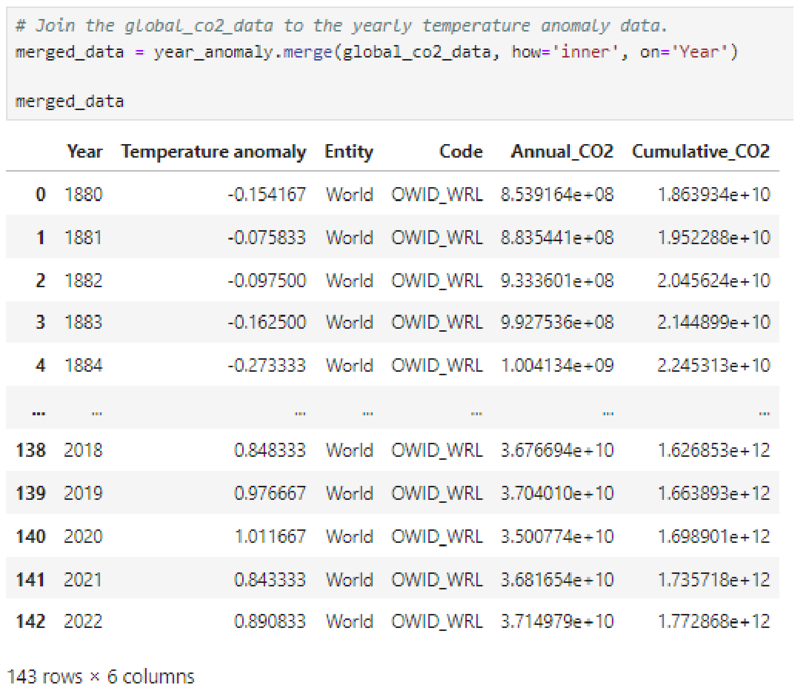
In many scenarios, you might find that you have data you'd like to *attach* to each other from different tables or datasets. In this case, we want to attach data about the global temperature anomaly to data about CO_2 production. These bits of data are sitting in different tables, so we're going to need to take data from one and connect it to the other.

To do this, we need to find a way of matching records from one table with records in the other. In this case we need a column which tells us *how* to match the entries together. This column is called the **key**, and tells us the relationship between the two tables. In our case, the **key** is the **year**, as this reflects the relationship between the entries in `year_anomaly` and `global_co2_data`.

There are some different kinds of joins. There are **inner**, **outer**, **left** and **right** joins, but they all achieve the same goal - combining data from different tables.

The join we're going to perform is the **inner join**. In an inner join, the result of joining two tables on a given **key** is a new table which contains only data which corresponds to a given key in both tables (e.g. both the CO_2 data and the temperature data must both have an entry for 1980 for the 1980 entry to appear in the resulting table. Only those key values which appear in both tables will appear in the output.

To do an inner join with pandas, we use the `merge` method of a DataFrame.



```
# Join the global_co2_data to the yearly temperature anomaly data.
merged_data = year_anomaly.merge(global_co2_data, how='inner', on='Year')

merged_data
```

	Year	Temperature anomaly	Entity	Code	Annual_CO2	Cumulative_CO2
0	1880	-0.154167	World	OWID_WRL	8.539164e+08	1.863934e+10
1	1881	-0.075833	World	OWID_WRL	8.835441e+08	1.952288e+10
2	1882	-0.097500	World	OWID_WRL	9.333601e+08	2.045624e+10
3	1883	-0.162500	World	OWID_WRL	9.927536e+08	2.144899e+10
4	1884	-0.273333	World	OWID_WRL	1.004134e+09	2.245313e+10
...	...	...	...	...	...	...
138	2018	0.848333	World	OWID_WRL	3.676694e+10	1.626853e+12
139	2019	0.976667	World	OWID_WRL	3.704010e+10	1.663893e+12
140	2020	1.011667	World	OWID_WRL	3.500774e+10	1.698901e+12
141	2021	0.843333	World	OWID_WRL	3.681654e+10	1.735718e+12
142	2022	0.890833	World	OWID_WRL	3.714979e+10	1.772868e+12

143 rows × 6 columns

The first argument is the DataFrame we wish to connect, `how` is the kind of join, and `on` gives the key. In this case, we're joining the `global_co2_data` to the `year_anomaly` data on the `Year` column.

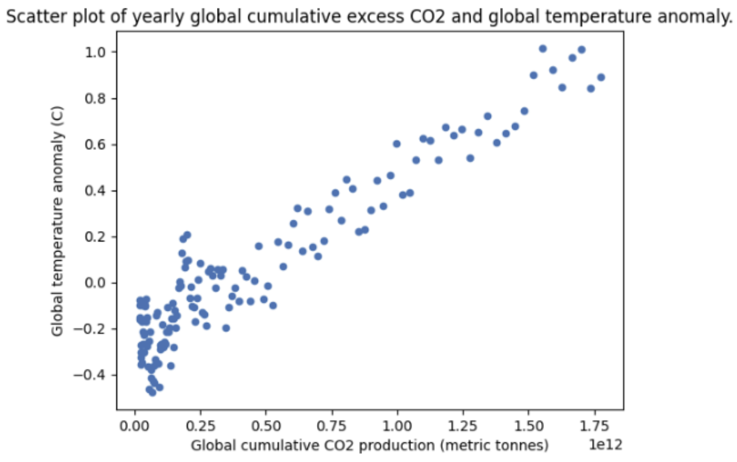
As you can see, what we get is a new table which contains all of the data from both tables where they are both specified for a given year. In practice, this is every year between 1880 and the year 2022. This is great! We now have a table of all global data containing both CO_2 accumulation and the temperature anomaly. Let's see what it looks like!

6.5 Plotting the Data

We're going to use the native pandas plotting method to start with, though we could just as easily use `matplotlib`. To do this, we take the DataFrame with the data and use the `.plot` method. What we get is this unsettling graph, clearly showing the relationship between CO_2 accumulated in the atmosphere and the global temperature anomaly.

```
# Plot CO2 against temperature anomaly using pandas.
merged_data.plot(kind = "scatter",
                x = "Cumulative_CO2",
                y = "Temperature anomaly",
                xlabel = "Global cumulative CO2 production (metric tonnes)",
                ylabel = "Global temperature anomaly (C)",
                title = "Scatter plot of yearly global cumulative excess CO2 and global temperature anomaly.")
```

```
<Axes: title={'center': 'Scatter plot of yearly global cumulative excess CO2 and global temperature anomaly.'}, xlabel='Global cumulative CO2 production
(metric tonnes)', ylabel='Global temperature anomaly (C)'>
```



You can see that there are a larger number of points down in the bottom left corner. This tells us that there were lots of years in the record where the global cumulative CO_2 production was still incredibly low, before the modern world got underway.

6.6 Fitting a line of best fit

Let's finish by trying to calculate a line of best fit for this graph. To do this we're again going to use the `scipy.stats` module to compute the slope and intercept of a line of best fit, which is computed using linear regression.

When we run the regression with `scipy.stats.linregress`, the result is an instance of the `LinregressResult` object, which contains the slope and intercept data as **attributes**. We've used attributes implicitly in this course before - they're simply variables stored in an object, and you can access them with `object.attribute`. We then round these, and see what the result is.

```
# Import scipy.
import scipy

# Perform a linear regression on the data.
regression = scipy.stats.linregress(x = merged_data['Cumulative_CO2'], y = merged_data['Temperature anomaly'])

# Get the slope and the intercept.
slope = np.round(regression.slope, 15)
intercept = np.round(regression.intercept, 2)
# Print out the values.
print("The slope of the line is " + str(slope) + "\nand the intercept is " + str(intercept) + ".")
```

```
The slope of the line is 7.14e-13
and the intercept is -0.27.
```

Another attribute saved in the `LinregressResult` object is the p-value associated with the following two hypotheses:

H_0 The slope of the regression is zero.

H_1 The slope of the regression is not zero.

In particular, the chances of observing this data if the line of best fit should actually be flat is given by the p-value. In this case:

```
# Print the p-value.
print("The probability that there is no linear relationship given the data is " + str(regression.pvalue) + ".")
```

```
The probability that there is no linear relationship given the data is 8.356385542716155e-73.
```

That's 72 zeros after the decimal point, so we can be pretty sure that the relationship exists. But how was this calculated? We'll come to that in the next lecture, where we'll take your statistical modelling skills to the next level!

7 Simulation and Hacker Statistics

It's an unfortunate truth about the world that reality is complicated. It routinely refuses to match analytical[5] models exactly, and this can make reasoning about data and relationships quite complicated. In practice, this is often due to the fact that it can be challenging to isolate the underlying distribution in the data with a formula - many empirical distributions get quite messy quite quickly.

Consider, for example, how earlier in the course we considered the distribution of BMI. Since this is computed with

$$\text{BMI} = \frac{\text{Weight (kg)}}{(\text{Height (m)})^2},$$

we have no reason to believe that the BMI might have a normal distribution. In fact, it has a tendency to skew upwards, with a slightly longer tail on higher BMIs than lower BMIs. The expected analytical form of the distribution of BMI is not incredibly straightforward to discern.

Thus far, we have only seen the standard Gaussian distribution. We aren't going to go into depth into any other kinds of distributions for lack of space (although there are a good number and they're very interesting), but instead we're going to try and give you the skills you need to be able to work in situations where the distribution is not obvious or analytical.

7.1 Length of movies on Netflix

To get a feel for an unusual distribution, we're going to take some real data from Netflix and look at the lengths of the movies which are provided on the platform. It's going to take a few steps, and this time we're importing from an excel file (`.xlsx`). We start by importing pandas and scipy, as we're going to need scipy to look at the distribution.

```
# Import some packages.
import scipy
import pandas as pd

# Load the netflix dataset.
netflix = pd.read_excel('netflix_titles.xlsx')
```

Here we used the pandas function `read_excel` which can import data from excel spreadsheets. In order to get this function to work, you might also have to install a package (with pip as before) called `openpyxl`. This is an **optional dependency** for pandas, meaning that while some parts of the library use this package, it won't install it for you.

Looking at the netflix DataFrame, we see that it has a column called **type** and a column called **duration**. We'll select the movies which are on Netflix by filtering on the `type` column.

```
# Select the movies from the data using .loc for safe indexing.
movies = netflix.loc[netflix['type'] == 'Movie'].copy()
```

While it's quite possible to use `netflix[netflix['type'] == 'Movie']`, we saw previously that this will sometimes raise a warning, because Pandas is scared you might confuse a **view** for a **copy**. Roughly speaking, a **view** is a representation of the data in the DataFrame, accessing the same locations in computer memory. A **copy** creates a new version of the data, stored at a different location in the computer memory. To make it very explicit to Pandas that we want to look at data in the original table, we use `.loc` (which takes either Boolean series as here, or it can take a slice of rows and then a slice of columns). See the documentation at `https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.loc.html` for more information on .loc.

To be explicit that we then want to take this data and make a copy, we use `.copy()`, which will save this data to a new location in memory. Taking the time to do this extra step makes sure that we don't accidentally change something we don't intend to (although often this problem doesn't arise).

[5]**Analytical** in this context means that there's a mathematical formalism which describes the situation and allows us to compute the exact form of the solution.

Another thing you might notice if you look at the data is that all of the durations are stored in the form of a string like "90 min" or "101 min". We'll need to format this so that this data is instead stored as an integer.

```
# Create a new column dur_min.
movies['dur_min'] = movies['duration'].apply(lambda dur_text: str(dur_text).split()[0])
```

There are a few more things to explain here. Firstly we select the `duration` column in the movies DataFrame. We're using the method `.apply` which takes, as an argument, a function, which it will then apply to every entry in the series. The function we've given doesn't look like anything we've seen, but it's relatively straightforward notation. It's interesting though, so worth a brief interlude.

7.1.1 Lambda functions

A **lambda function** is a function which is short enough that it doesn't need a name. In many scenarios you might have a short function that you want to apply but you don't want to spend additional time defining the function, and this is when lambda functions are most useful. The syntax is like this:

```
lambda (arguments): (output in terms of arguments)

# A function which squares all inputs.
lambda x: x**2

# A function which adds together its two arguments.
lambda x, y: x + y

# A function which just returns an uppercase version of a string.
lambda word: word.upper()
```

What our lambda function does is take the entries like "100 min" (here designated as `dur_text`), and makes sure they're actually a string with `str`. All strings have a useful method called `split`, which turns a string into a list of words. We then index that list to get the first word, so we get a string corresponding to "100".

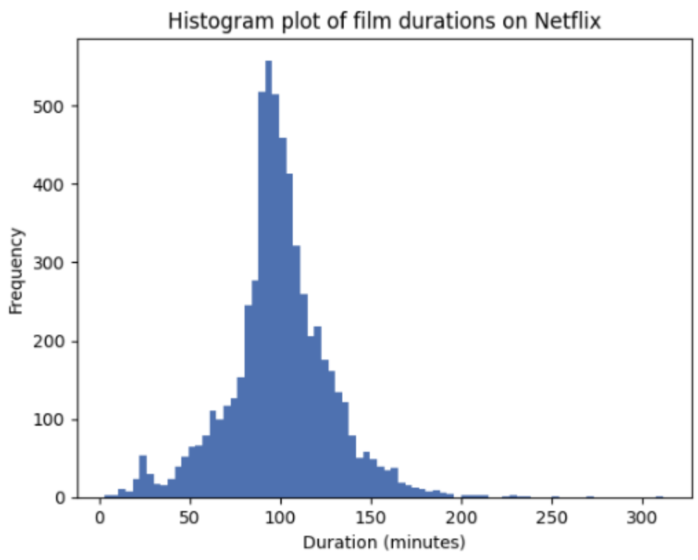
We could try to immediately convert it to an integer too, but this will actually give an error. Looking at the data, we see that some of the durations are given as 'nan'. This is because some of the movies are included without reference to a length - possibly because they made use of Netflix's "interactive storytelling" feature. You might have seen this if you've seen Black Mirror's **Bandersnatch**, or the life-changing phenomenon of **Spirit Riding Free: Ride Along Adventure**. For the sake of simplicity, we'll remove these movies and then convert the durations on the remaining movies to an integer.

```
# Remove the nan entries using .loc
movies = movies.loc[movies['dur_min'] != 'nan']

# Change the data type to integers using .loc
movies['dur_min'] = movies['dur_min'].apply(lambda x: int(x))
```

With that in hand, we can finally plot a histogram of the data, to get a rough idea of the distribution of the movie lengths.

```
# Plot the durations in a histogram.
movies['dur_min'].plot(kind='hist', bins=80, xlabel='Duration (minutes)', title='Histogram plot of film durations on Netflix')
```

```
<Axes: title={'center': 'Histogram plot of film durations on Netflix'}, xlabel='Duration (minutes)', ylabel='Frequency'>
```



What we see is a broad spread, with a small peak around the 20-25 minute range for some short films and a larger peak around 90-100 minutes. This distribution is definitely not one we could analytically derive. Using the `.describe` method on the series gives us some summary statistics for the dataset.

```
# Get some basic statistics.
movies['dur_min'].describe()
```

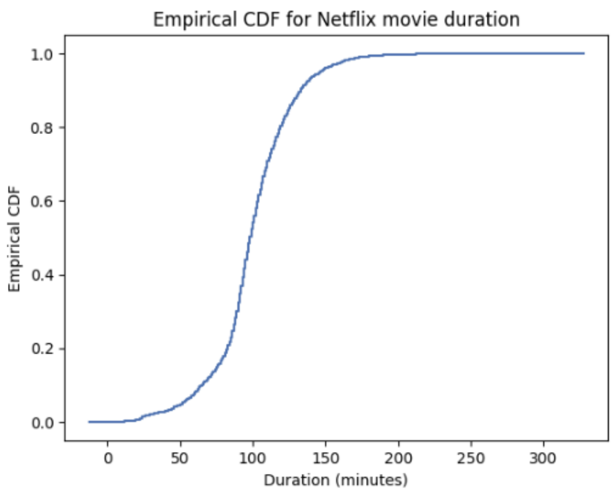
```
count    6129.000000
mean       99.578887
std        28.288598
min         3.000000
25%        87.000000
50%        98.000000
75%       114.000000
max       312.000000
Name: dur_min, dtype: float64
```

We see that there are 6,129 entries remaining in the data, after we removed TV Series and those movies with 'nan' lengths. The average movie, including short films, is about 99 and a half minutes long. The longest movie is a whopping 312 minutes! That's more than 5 hours!

To get a better idea of the distribution of these film lengths, we can also plot an empirical **cumulative distribution function**. This is a function which shows, given a value x for the length of a film, the probability that a randomly selected film would have a length less than x. To do this, we're going to use `scipy.stats.ecdf`.

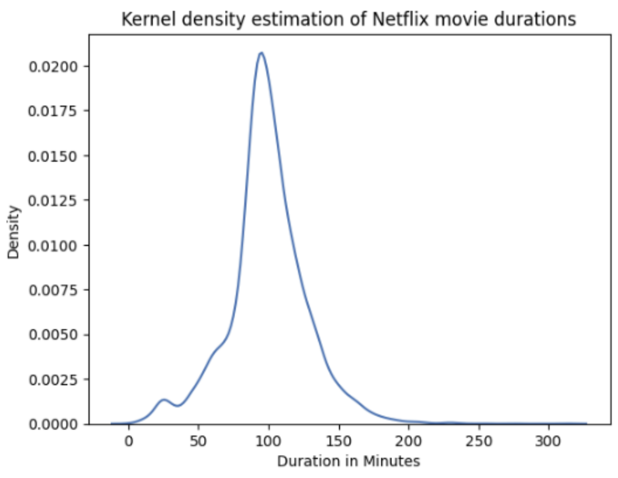
Don't worry too much about the the plotting code. Feel free to use this snippet of code directly to produce the empirical cumulative distribution function of the movie duration data.

```python
# Import scipy and matplotlib.
import scipy
import matplotlib.pyplot as plt

# Create an empirical distribution object.
cumulative_duration = scipy.stats.ecdf(movies['dur_min'])

# Plot the result.
ax = plt.subplot()
cumulative_duration.cdf.plot(ax)
ax.set_xlabel('Duration (minutes)')
ax.set_ylabel('Empirical CDF')
ax.set_title('Empirical CDF for Netflix movie duration')
plt.show()
```



Normally in statistics we work with probability density functions, **pdf**s, rather than cumulative distribution functions. There's a way to estimate this in python, but it's not quite as reliable as the empirical cdf. You have to make some assumptions about the smoothness of the distribution. Regardless, if you want to do it, you can use the `kdeplot` from the `seaborn` package to do so.

```python
import seaborn as sns
import matplotlib.pyplot as plt

# Estimate the pdf.
sns.kdeplot(data=movies, x='dur_min')

# Add an x-label and a title.
plt.xlabel('Duration in Minutes')
plt.title('Kernel density estimation of Netflix movie durations')

# Show the plot
plt.show()
```



7.2 Monte Carlo Methods

Loosely speaking, **Monte Carlo methods** are a class of techniques which depend upon repeated computational sampling and simulation to gain insight about a system. Many systems are much too complex or random to reasonably make good deductions about. However, running a simulation over and over again, you might get a good idea about what, on average, you might expect to happen. These methods are named after the Casino Monte Carlo.

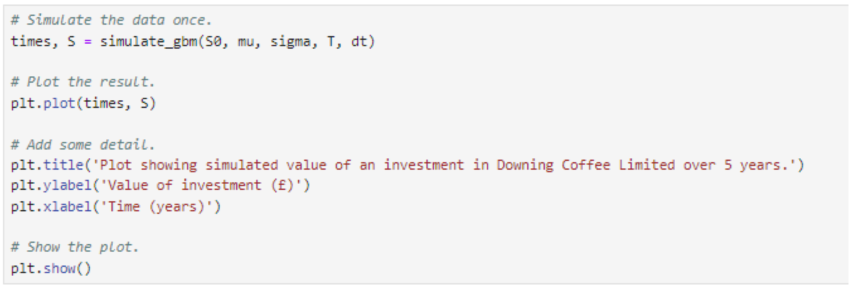
Performing simulations with python can make some very complex scenarios relatively straightforward to simulate. To demonstrate this, we're going run a stock market simulation on a single stock - that of **Downing Coffee Limited**. We're trying to decide whether or not we should invest our well earned 100 pounds into this high-volatility coffee stock. To help us make our decision, we create a model which simulates the stock price of Downing Coffee Limited. Suppose that our model is absolutely fantastic - its simulations are incredibly realistic, but we can never truly predict the future, so we always have to account for random noise. Here's the code that runs our model.

```
def simulate_gbm(S0, mu, sigma, T, dt):
    """
    Simulates a Geometric Brownian Motion path.

    Parameters:
    S0 (float): Initial stock price
    mu (float): Expected return
    sigma (float): Volatility of the stock
    T (int): Time horizon
    dt (float): Time step size

    Returns:
    numpy.ndarray: Simulated GBM path
    """
    n_steps = int(T / dt)
    times = np.linspace(0, T, n_steps)
    S = np.zeros(n_steps)
    S[0] = S0
    for t in range(1, n_steps):
        Z = np.random.normal(0, 1)
        S[t] = S[t-1] * np.exp((mu - 0.5 * sigma**2) * dt + sigma * np.sqrt(dt) * Z)
    return times, S

# Parameters
S0 = 100  # Initial stock price
mu = 0.03  # Drift coefficient
sigma = 0.05  # Volatility
T = 5  # Time horizon (5 years)
dt = 0.01  # Time step size

# Simulate GBM
times, S = simulate_gbm(S0, mu, sigma, T, dt)
```

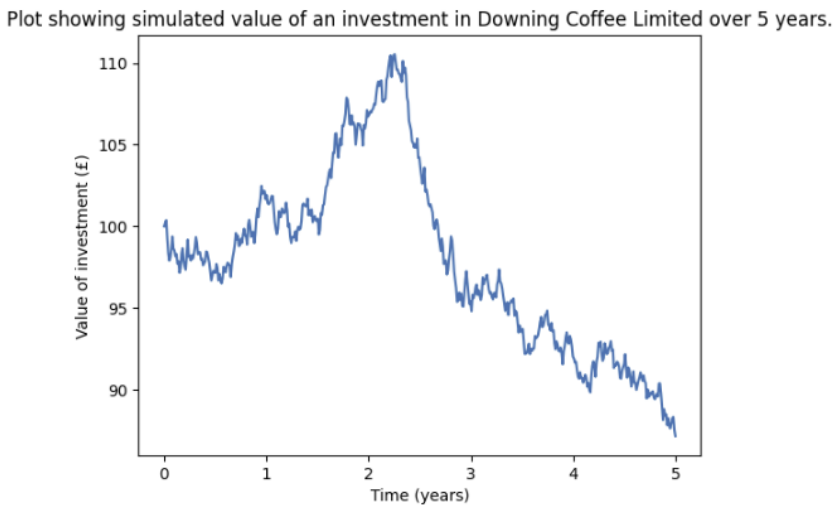
Running the function `simulate_gbm` will give us back two numpy arrays. The `times` array contains time points across 5 years. The `S` numpy array consists of the stock price at each time point. Plotting these against each other shows the behaviour of the value of the stock over 5 years.

The model we're using is based on **geometric brownian motion**, which actually captures the characteristic *flickering* movements of many investments. That isn't to say that it will predict the movement of the market perfectly - far from it, but it does look pretty good.

Let's run our simulation once to see one possible outcome for investing our 100 pounds into Downing Coffee Limited.



```python
# Simulate the data once.
times, S = simulate_gbm(S0, mu, sigma, T, dt)

# Plot the result.
plt.plot(times, S)

# Add some detail.
plt.title('Plot showing simulated value of an investment in Downing Coffee Limited over 5 years.')
plt.ylabel('Value of investment (£)')
plt.xlabel('Time (years)')

# Show the plot.
plt.show()
```

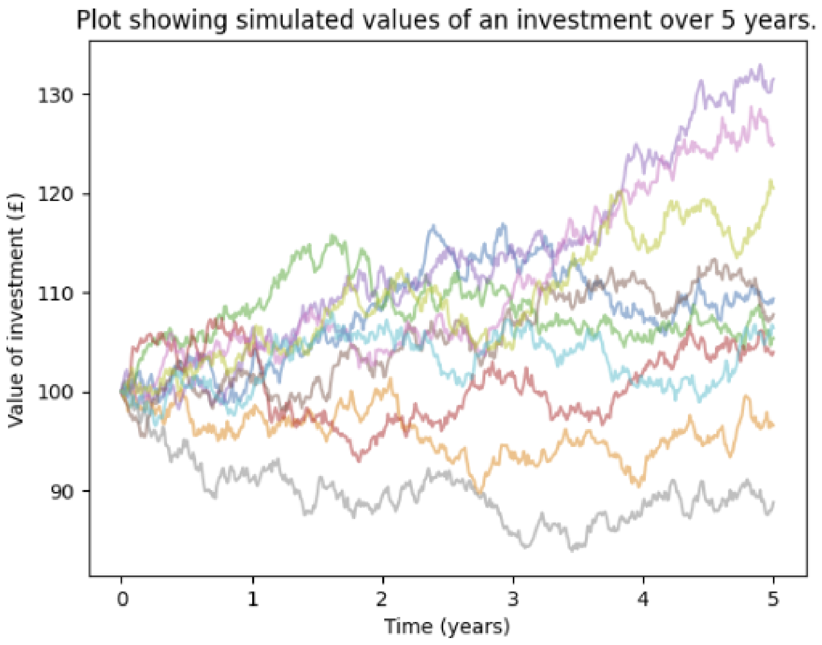


Well that doesn't look very good. In fact, the final value of our investment in this case was 85 pounds and 15 pence. Not a stellar performance on this high-volatility stock. Does this mean that we shouldn't invest in Downing Coffee Limited? Not necessarily - what if we simply had some bad luck and didn't see what could have been?

Let's try re-running the simulation 10 times, and see what happens each time.

```
# Initialise empty list of values.
final_values = []

# Specify the number of times to run the simulation.
runs = 10

# Run the loop
for run in range(0, runs):
    times, S = simulate_gbm(S0, mu, sigma, T, dt)
    final_values.append(S[-1])
    plt.plot(times, S, alpha = 0.5)

plt.title('Plot showing simulated values of an investment over 5 years.')
plt.ylabel('Value of investment (£)')
plt.xlabel('Time (years)')
```

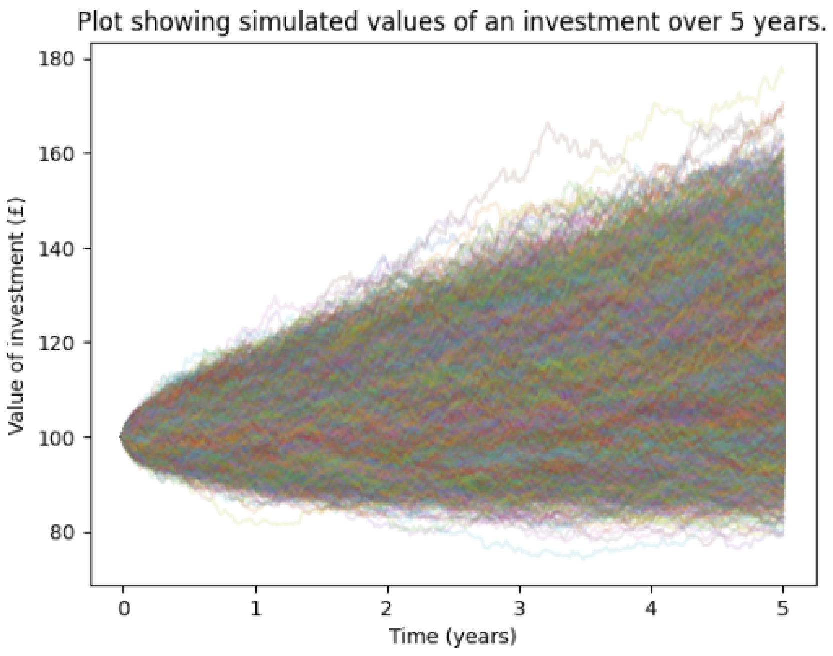
```
Text(0.5, 0, 'Time (years)')
```



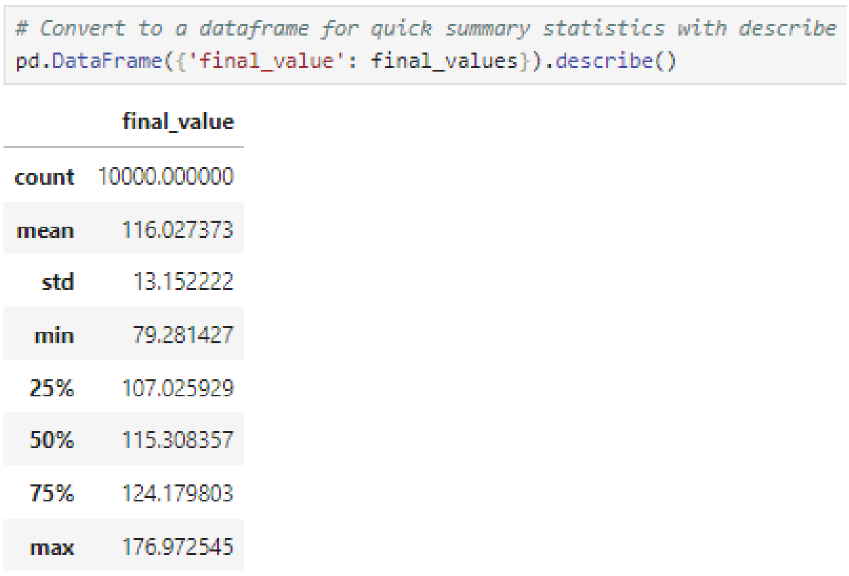
Looks like we might have just had a bad run. In this case, eight out of ten of the simulations **increased** the value of our investment. But ideally we'd like to be even more precise than this. What's the probability that our investment is going to appreciate in value? What is the expected (average) return on our investment? To do this we're going to run the simulation 10,000 times, and see what we get out the other side.

```
# Initialise empty list of values.
final_values = []

# Specify the number of times to run the simulation.
runs = 10000

# Run the loop
for run in range(0, runs):
    times, S = simulate_gbm(S0, mu, sigma, T, dt)
    final_values.append(S[-1])
    plt.plot(times, S, alpha = 0.15)

plt.title('Plot showing simulated values of an investment over 5 years.')
plt.ylabel('Value of investment (£)')
plt.xlabel('Time (years)')
```

```
Text(0.5, 0, 'Time (years)')
```



Now we've run this simulation 10,000 times, we can see that in the vast majority of simulations the value of the investment actually increased. To quickly see some summary statistics on the data, we can put it into a pandas DataFrame and then use the `describe` method to get the following:



```
# Convert to a dataframe for quick summary statistics with describe
pd.DataFrame({'final_value': final_values}).describe()
```

	final_value
count	10000.000000
mean	116.027373
std	13.152222
min	79.281427
25%	107.025929
50%	115.308357
75%	124.179803
max	176.972545

From this we see that the average return on investment was 16 pounds and 3 pence. The standard deviation was about 13 pounds, and the median return was 15 pounds 30 pence. What percentage of the runs resulted in a loss? To see this we can look up the percentile corresponding to the final value 100 using `scipy`.

```
# Get the percentile corresponding to £100
scipy.stats.percentileofscore(final_values, 100)

>>> 10.43
```

This value varies a small amount, so we'll say that roughly 10% of the simulated investments actually made a loss over the 5 year period. Interesting stuff! Would you consider investing in **Downing Coffee Limited**, given this new information?

7.3 Hacker statistics

When we apply Monte Carlo methods to statistics, we're doing what a lot of data scientists might refer to as **hacker statistics**. For example, we made an estimation of the probability that we would have a positive return on our investment, rather than making a loss.

We saw that, on average, if the model is simulating the stochastic behaviour of the market perfectly, that we would always expect to see an average return on investment of 16 pounds over 5 years. But how confident can we be in that number?

7.3.1 Confidence intervals

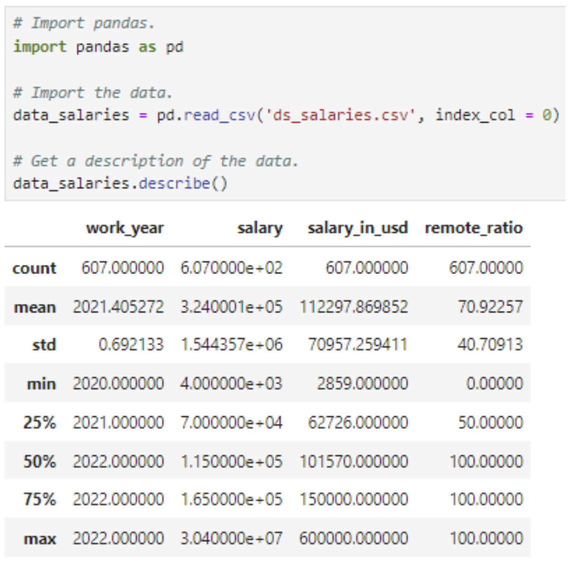
Earlier when we were looking at hypothesis testing, we used **p-values** to check whether or not our hypothesis is correct. In that scenario we had two hypotheses, the null hypothesis and the alternative hypothesis, and the p-value was the probability that we incorrectly dismissed the null hypothesis and accepted the alternative hypothesis.

In the same vein, we can ask ourselves what the probability is that a certain measurement we have taken of a population is correct within a reasonable margin of error. When we do this we extract two numbers - an upper and a lower bound, and the probability that the population statistic lies in between these bounds will then be equal to a pre-specified probability (the confidence level).

To one way to compute confidence intervals on sample statistics, we're going to use a powerful method called **bootstrapping** to look at the distribution of salaries of data scientists.

7.3.2 Bootstrapping

The dataset we're going to use consists of a random sample of 608 different salaries of individuals working in data science across different countries, along with other bits of information about the size of their company and official title. This dataset, like some of the others in this course, is taken from **Kaggle** (`https://www.kaggle.com/datasets/sanyacodes/salaries-in-data-science`).



```python
# Import pandas.
import pandas as pd

# Import the data.
data_salaries = pd.read_csv('ds_salaries.csv', index_col = 0)

# Get a description of the data.
data_salaries.describe()
```

	work_year	salary	salary_in_usd	remote_ratio
count	607.000000	6.070000e+02	607.000000	607.00000
mean	2021.405272	3.240001e+05	112297.869852	70.92257
std	0.692133	1.544357e+06	70957.259411	40.70913
min	2020.000000	4.000000e+03	2859.000000	0.00000
25%	2021.000000	7.000000e+04	62726.000000	50.00000
50%	2022.000000	1.150000e+05	101570.000000	100.00000
75%	2022.000000	1.650000e+05	150000.000000	100.00000
max	2022.000000	3.040000e+07	600000.000000	100.00000

Here we import the data and immediately use the `describe` method to get an idea of the distribution of the data. In particular, we see that the mean salary observed in our sample in USD is $112,298. But how much can we rely on this number? It might be that, again, due to chance, our observed mean might be slightly off of the actual value for the entire world's population of data scientists. We're going to compute a confidence

There are broadly two approaches to making up for the difference between a sample and a population. The first is to make a model of the population with some assumptions you might reasonably believe to be true; for example, that it has a normal or Gaussian distribution. Alternatively you can capitalise on the information you already have from the sample. Bootstrapping is a method of achieving the second.

When we bootstrap, we are taking a new sample from the one we already have, except we sample with replacement. For example, given an original sample:

$$1, 1, 2, 3, 6, 7, 8, 10, 10, 11$$

we could extract the following resample:

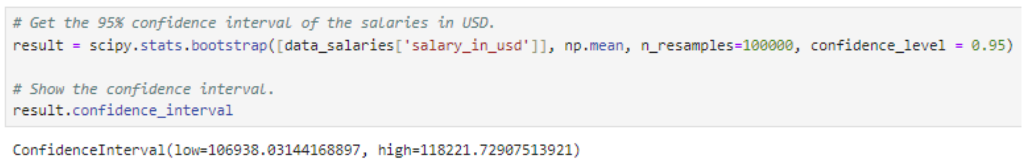
$$1, 2, 3, 6, 6, 7, 10, 10, 10, 11$$

Note that in our resample we might have several copies of original data. The result is that every observation in the resampling is drawn from the distribution of the sample (which is the only distribution we reasonably are able to assume looks like the population distribution).

If we then compute statistics on this new group, e.g. we compute a mean or standard deviation or some other thing we're trying to compute, then with maximum likelihood it will look as if we had simply taken a sample from the whole population. Repeating this resampling process over and over again will give us effectively a family of different possible samples, and for each one we will compute the test statistic, giving us a distribution of that statistic.

In essence, sample distributions are a good estimate for a population distribution, so we can use it to deduce things about the population distribution.

To do this in python, we can again use `scipy.stats` to take the salary data and compute the mean on 100,000 resamples. We're going to do this at the 95% confidence level, meaning that there's a 95% chance that the population mean lies within the range we find.



```
# Get the 95% confidence interval of the salaries in USD.
result = scipy.stats.bootstrap([data_salaries['salary_in_usd']], np.mean, n_resamples=100000, confidence_level = 0.95)

# Show the confidence interval.
result.confidence_interval
```

```
ConfidenceInterval(low=106938.03144168897, high=118221.72907513921)
```

To sum up our findings, there is a 95% chance that the mean salary (in USD) of data scientists worldwide lies in the range of $106,938 to $118,222. If we had more data, we could narrow this range even further.

Bootstrapping is a powerful technique for making deductions about population statistic distributions based off of samples of a population. They're very common in science for finding confidence intervals, where it's not always immediately straightforward to model the population statistic. It's a useful tool to remember!

8 Machine Learning: A Brief Introduction

8.1 Introduction

In the modern world it's very likely you've come across the phrase **artificial intelligence**. This notion, whereby a machine acts with some degree of intelligence, is surprisingly outdated. Artificial intelligence in the old school sense can include, for example, an agent which is given a series of **if** x **then** y rules about how it should behave. With sufficiently many of these rules, a machine can give a relatively thin impression of being intelligent, with the caveat that it can never respond to new situations appropriately - it cannot take what knowledge it has and generalise with it, or use it to perform more complex behaviour.

Because of this reason, modern research in AI is focused on the much more powerful realm of **Machine Learning** or **ML**. Machines can have useful abilities with old-school AI, but machine learning can give machines abilities that old-style AI never could have. The key difference is that in machine learning, the machine learns patterns in the data itself. That way, rather than trying to list every possible piece of information that might inform a decision, the machine can deal with this part for you.

8.2 Agents and Models

In AI, there are broadly two classes of entities that we might come across. The first such entity, on which we're going to focus for the rest of this lecture, are **models**. A **model** is any kind of system which is designed to make predictions. It fundamentally takes input from data which is known and tries to predict something which we don't already know. For example, if I had a good amount of Google street view data, I could try and develop a model to predict the price of a house based on its facade facing the street. The model doesn't do anything else.

In contrast, an **agent** is an entity which takes inputs from some kind of an environment and performs actions (often called policies in ML) based on its inputs. This idea is actually incredibly general, and it appears all over science and economics. For example, countries act like agents, as do the people that live there. They all experience some kind of an environment (be it the global political and economic climate, or your living room) and take actions (like increasing interest rates, or reading a book). Agents often incorporate models in order to help them take actions.

The holy grail of AI research is a kind of agent with enough intelligence that it can perform most (if not all) human tasks as well as humans can. When this occurs, the agent is said to possess **artificial general intelligence**, or **AGI**. Further down the line, if there exists an agent which is strictly **outperforming** humans on most or all tasks, then this is called **artificial super intelligence**, or **ASI**.

Both AGI and ASI have the potential to make remarkable changes to the global landscape, and, if AI scientists are to be believed, seem to be getting closer and closer. About 5 years ago, only 45% of scientists expected AGI to occur before 2060. Today, nearly half of scientists believe it's going to happen before 2048. Some people, such as Sam Altman, CEO of OpenAI, think it'll be achieved within 5 years. In any case, scientists are taking the possibility quite seriously, so don't be surprised if you find yourself interacting with agents with increasing frequency.

8.3 Types of machine learning

Broadly, machine learning scientists use three types of methods to help develop a model or an agent. These are **supervised** learning, **unsupervised** learning and **reinforcement** learning.

8.3.1 Supervised learning

When you teach a human how to do something, especially if it's completely new or unfamiliar, then you're usually inclined to give them lots of different examples. This method is also applied when training machines. In essence, a supervised learning algorithm will provide the machine with lots of different examples about what kind of answer it's supposed to give. For the street-view house-price prediction engine, for instance, we might give the machine lots of pictures of different houses and tell it what the corresponding house price was. Usually in supervised learning, the model looks at the input data and then makes a prediction (even if this is completely wrong). Then, when provided with the actual answer, it will adjust itself so that in future cases it will be closer the correct answer. The main problem with supervised learning is that it requires large, labelled datasets, which can often be expensive or difficult to produce. We'll look at two supervised learning techniques: **linear regression** and **logistic regression**, although many more exist (e.g. artificial neural networks).

8.3.2 Unsupervised learning

Given the difficulty of producing large labelled datasets, you might wonder if machines can produce interesting results without being provided with labels. These kinds of approaches, where the output of the machine is not explicitly corrected, are called **unsupervised learning** methods. In these methods, the machine might be breaking up the input data into different clusters, or breaking them down into different components. We're going to have a look at one clustering algorithm (called **k-means clustering**) and one dimensionality reduction (**principal component analysis**).

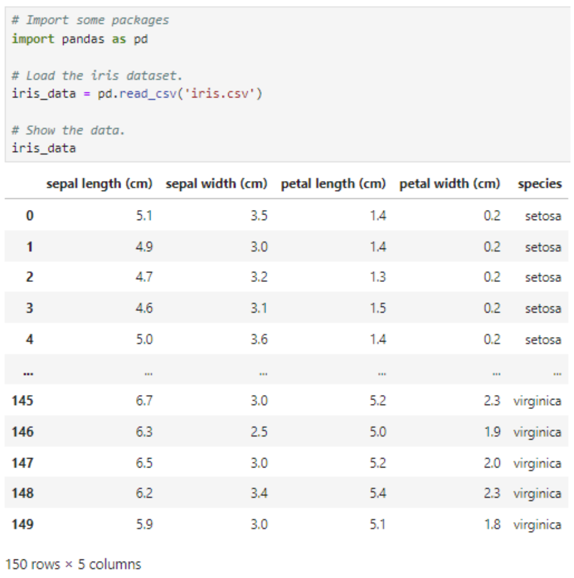
8.3.3 Reinforcement learning

This one's a little bit different from the other two. Reinforcement learning is when an agent takes inputs from an environment and acts in that environment to maximise a certain score or goal. For example, suppose you want to train an agent to play a video game, and to do this you give it the goal of maximising it's percentage progress through the game. In this case, the agent will act so as to maximise its progress and change its behaviour when it starts over each time to try and make it progress even further. We won't look at these kinds of techniques any further, but they're incredibly interesting. If you're interested, there's lots of material on these kinds of models online[6].

8.4 Scikit-Learn

One of the best tools for implementing basic machine learning in python is the package **scikit-learn**. This package provides lots of relatively straightforward tools for easy access to machine learning techniques. To install it, navigate to your command line and run `pip scikit-learn`. It works well with data from NumPy and Pandas!

Let's load some data using Pandas. This is a classic dataset called the **iris** dataset, and it contains data about three different species of the genus *iris*.

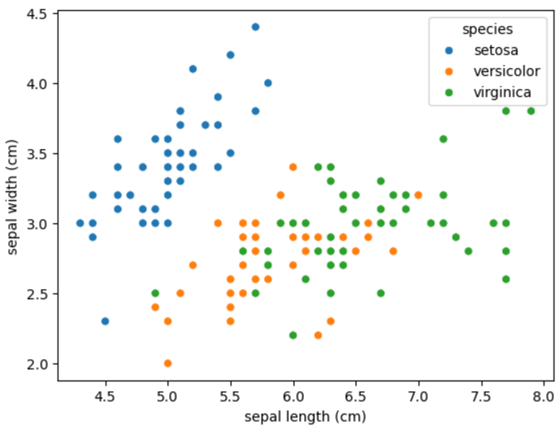


```python
# Import some packages
import pandas as pd

# Load the iris dataset.
iris_data = pd.read_csv('iris.csv')

# Show the data.
iris_data
```

	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)	species
0	5.1	3.5	1.4	0.2	setosa
1	4.9	3.0	1.4	0.2	setosa
2	4.7	3.2	1.3	0.2	setosa
3	4.6	3.1	1.5	0.2	setosa
4	5.0	3.6	1.4	0.2	setosa
...	...	...	...	...	...
145	6.7	3.0	5.2	2.3	virginica
146	6.3	2.5	5.0	1.9	virginica
147	6.5	3.0	5.2	2.0	virginica
148	6.2	3.4	5.4	2.3	virginica
149	5.9	3.0	5.1	1.8	virginica

150 rows × 5 columns

[6]For example, see AlphaGo, AlphaZero, MuZero from Google.



We're going to use this data to try and predict the species using all of the other available data about the petals and sepals. For this reason, it's worth a quick detour to talk about features and targets.

8.4.1 Features and targets

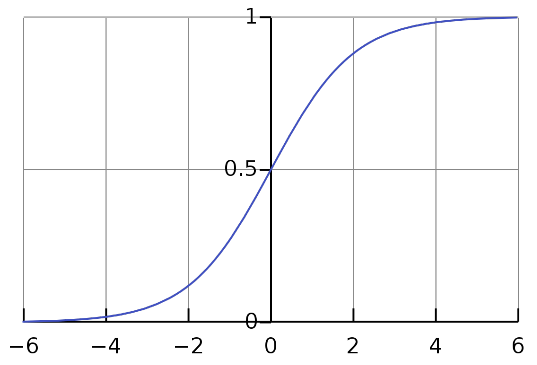
When we're building a model in ML, we want to make the separation between our input and output data quite explicit. When the model is making its prediction, the only data it should see are pieces of data called **features**. A **feature** roughly corresponds to a column in a table, or a certain type of information about the input. In our house-price prediction model, this might be images we provide, or an explicit calculation of the number of windows or number of garages, but not the price of the house, as the model won't see this when it makes its predictions.

The thing our model is trying to predict is called the **target**. The target can be any kind of data, but broadly speaking, if the target is *categorical*, such as when we're trying to predict the species of the iris plant, then the model is called a **classifier**. If the target is *continuous*, e.g. the price of a house, then the model is called a **regressor**. Both classifiers and regressors are known as **estimators** in scikit-learn.

In scikit-learn, there are three useful methods for each instance of an estimator. First, there is the `fit` method. This method will train the given model on its training data. Then the `predict` method will take in unseen data and make predictions using the model. Lastly, you can `score` the model to see how well the model performs. The exact metric used will depend on the estimator, or you can specify your own scoring system.

8.4.2 Logistic regression

We're going to proceed by using a method called **logistic regression** to predict the species of each flower in the `iris` dataset. Despite its name, logistic regression is usually thought of as a classification method. It works by fitting a logistic curve to the data.



The equation of the basic logistic curve is

$$\sigma(x) = \frac{1}{1+e^{-x}}$$

and it varies between 0 and 1. Because of this property, sometimes the logistic function is viewed as representing an output *probability*, even though this isn't strictly a probability in the usual sense.

To each of the possible output classes, based on some numerical input data, the model will take a weighted sum of the inputs and try to compute the probability of each class compared to each other. Suppose, for example, that you had a series of features $X_1, \ldots, X_4$ and a target T. Then logistic regression will make an estimate of the probabilities that each observation belong to each class t in the target variable by computing something like

$$\sigma(a_{1t}X_1 + a_{2t}X_2 + a_{3t}X_3 + a_{4t}X_4)$$

and then scaling it against the probabilities of the other classes. The class with the highest probability is then selected as the output class from the model. If you want more details on the mathematics of Logistic Regression, you can find it online or a brief introduction is also given on this page from scikit-learn: `https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression`.

To implement this in scikit-learn, we're going to work through a few different steps. Firstly, we're going to import the code we're going to need from scikit-learn, take the data we imported earlier and split it into different parts.

```
# Import model and some utilities.
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay
import matplotlib.pyplot as plt

# Select the data.
X = iris_data[['sepal length (cm)', 'sepal width (cm)', 'petal length (cm)', 'petal width (cm)']]
y = iris_data['species']

# Split the data into training data and test data.
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2, stratify = y, random_state = 3)
```

In the first block we import the classifier object `LogisticRegression` and some other tools. By default, Scikit-learn won't import all of its code for you from its library, so if you want to use a particular piece of the code, you usually have to import it first. In the second block we specify our features and our target. When working with scikit-learn, it's common to label your features as X and your target as y.

When we work with machine learning models, it's incredibly important to **test the model on different data to where it was trained**. This is one of the most fundamental rules in ML - if you don't do this, then you'll find that your model might not be able to generalise to the real world at all; it could, for example, learn the solution to each of the training problems almost by rote and be unable to generalise. This issue, called **overfitting**, can be easily spotted if you separate data to train from data to test.

The useful function `train_test_split` will help us break our data into two parts. In this case, `test_size` is set to 0.2, so we'll keep 80% of our data for training purposes, and 20% for testing purposes. You'll also see `stratify = y`. This means that when it breaks the data into two groups (which it does at random), it will make sure that the ratio of different targets is preserved in the training and testing sets. If we didn't do this, we might accidentally reduce the number of different observations of a certain species in our test data, leading to somewhat more unreliable results. Lastly, we see a `random_state = 3` - this is a seed which enables you to replicate your data split, but doesn't always need to be specified.

Now for the fun stuff.

```python
# Initialise the classifier.
iris_model = LogisticRegression()

# Fit the model.
iris_model.fit(X_train, y_train)

# Make predictions.
y_pred = iris_model.predict(X_test)

# Score the model.
score = iris_model.score(X_test, y_test)
print("The model accuracy was " + str(score) + ".")
```

```
>>> The model accuracy was 0.9.
```

Firstly, we initialise an instance of our classifier. We'll name our classifier `iris_model` to keep it straightforward. When we call `LogisticRegression()`, we can also specify other parameters to change how the classifier behaves. In this instance, we've gone with the default behaviour for our classifier.

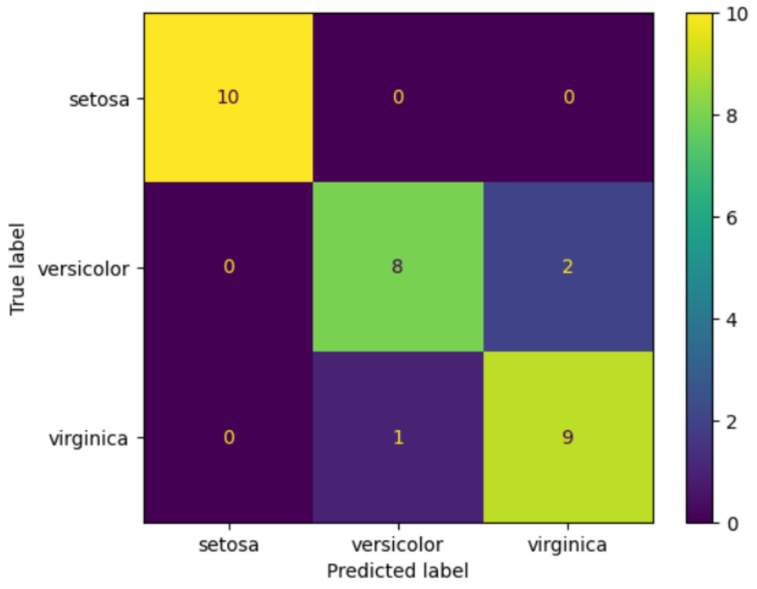
Now we can use the training data to fit our model. In scikit-learn, this is as simple as writing `.fit(X_train, y_train)`. This tells the model which features it should train on and which target to predict. Once we've used the `.fit()` method, we can then use the `.predict(X_test)` method to see what the model predicts should be the corresponding target values for the given test features. We've saved this, as with standard scikit-learn nomenclature as `y_pred`.

Lastly, we've obtained an accuracy score from the test data. Note that the `.score` method doesn't require that we use `.predict` first - we've just saved the predictions so we can see where our model was performing best in a minute. Score will perform the prediction for you. In this case, this model actually works well - 90% of its classifications were correct!

To see where the model might falter slightly, we're going to use the **confusion matrix** we imported earlier. This will be a matrix showing, for each class, what the model predicted compared to what the species actually was.

```python
# Make a confusion matrix.
conf_mat = confusion_matrix(y_test, y_pred)
ConfusionMatrixDisplay(conf_mat, display_labels=iris_model.classes_).plot()
plt.show()
```

```
The model accuracy was 0.9.
```



From this we can see that occasionally versicolor was predicted as being virginica - this happened twice, and once a virginica plant was predicted as being versicolor. Visualisations like this can really help make your models more transparent, making it easier to see where your models might be performing their best and their worst.

8.5 Linear Regression

We're now going to have a quick look at a regression problem using a linear regression model. This is one of the most commonly used models in all of machine learning, and they appear all over science in general, especially when we want to see if the effect we see in an experiment is due to and intervention or some other factor.

8.5.1 The broad mathematical details

When we fit a linear model to data, we consider a series of feature variables $X_1, \ldots, X_n$ and a given target variable Y. We assume that there is some kind of linear relationship between the feature data and the target data. That is, we assume that, roughly, there is a relationship

$$Y \approx a_1X_1 + a_2X_2 + \cdots + a_nX_n + b$$

where $a_1, \ldots, a_n$ are coefficients representing a contribution from each input variable, and b is the Y-intercept, the expected value of the target Y if all input variables were zero. However, even if we choose $a_1, \ldots, a_n$ and b perfectly, since our data is noisy, we don't expect that Y will be exactly equal to what's given on the right hand side. For that reason, we include a **residual** term, E, which makes up the difference. The residual is the variation in Y which cannot be explained linearly by $X_1, \ldots, X_n$. Hence we might write the model as

$$Y = a_1X_1 + a_2X_2 + \cdots + a_nX_n + b + E$$

where the equality is now exact.

This is exactly what linear regression is - it is any method which estimates the best values for $a_1, \ldots a_n$ and b so that the residuals E are as small as possible. The most commonly used method for estimating the coefficients is called **ordinary least squares** (OLS)[7], so much so that often people use *linear regression* and *ordinary least squares regression* interchangeably. Other methods for linear model regressions exist however, like **ridge** and **lasso** regressions, but we won't cover these here.

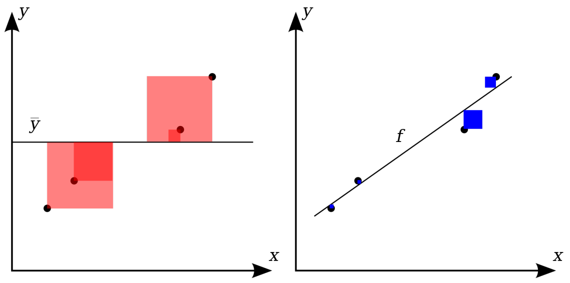


Figure 5: The total area of the squares (whose side length is given by the residuals E) is minimised so as to fit the line as closely as possible.

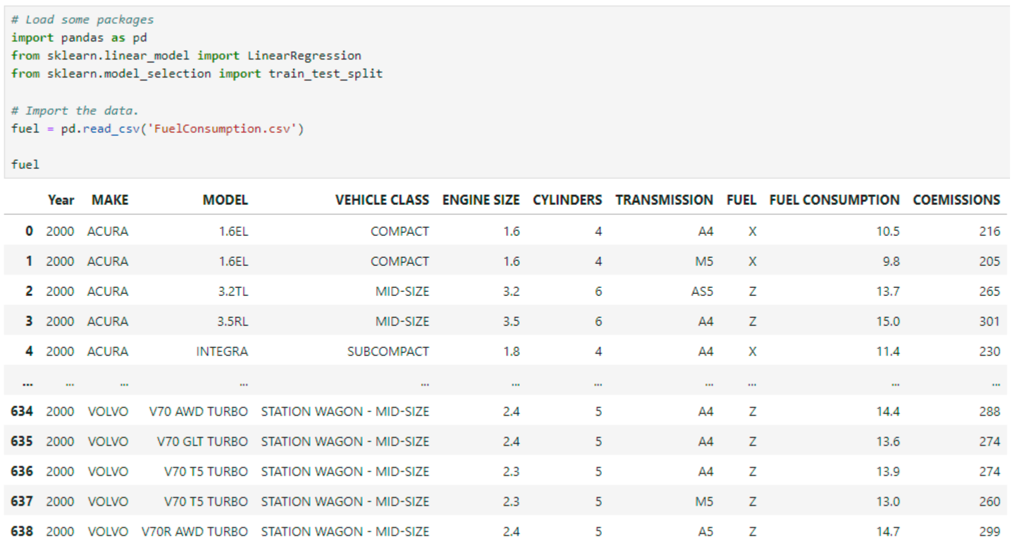
In OLS linear regression, the squared residuals are minimised through an optimisation algorithm, leaving us with a nice linear model which we can use to make predictions. This time, rather than making a classification, the model we've built takes numerical inputs and provides a numerical output. Let's see how this can be implemented in python.

[7]You might be wondering why we square the differences rather than just looking at the absolute value of the difference. One reason is that the absolute value, $|\cdot|$ is not nicely differentiable, making computations harder. The other reason is that squared differences fit more neatly into the rest of statistical theory, where standard deviation (also a squared difference) is nice and additive across independent variables - a property that $\sum|\cdot|$ would lack.

8.5.2 Linear regression in python

The data we're going to use consists of information about different cars, their brands, models, engines and fuel use, along with their CO_2 emissions. We're going to use the linear regression model to try to predict their carbon emissions based on their fuel consumption and the size of the engine.

To start, we'll import the data.

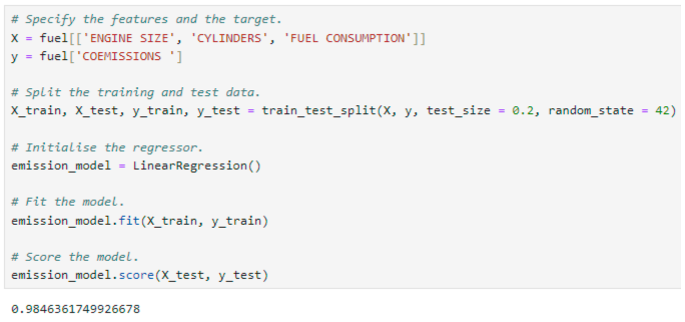


```python
# Load some packages
import pandas as pd
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import train_test_split

# Import the data.
fuel = pd.read_csv('FuelConsumption.csv')

fuel
```

	Year	MAKE	MODEL	VEHICLE CLASS	ENGINE SIZE	CYLINDERS	TRANSMISSION	FUEL	FUEL CONSUMPTION	COEMISSIONS
0	2000	ACURA	1.6EL	COMPACT	1.6	4	A4	X	10.5	216
1	2000	ACURA	1.6EL	COMPACT	1.6	4	M5	X	9.8	205
2	2000	ACURA	3.2TL	MID-SIZE	3.2	6	AS5	Z	13.7	265
3	2000	ACURA	3.5RL	MID-SIZE	3.5	6	A4	Z	15.0	301
4	2000	ACURA	INTEGRA	SUBCOMPACT	1.8	4	A4	X	11.4	230
...	...	...	...	...	...	...	...	...	...	...
634	2000	VOLVO	V70 AWD TURBO	STATION WAGON - MID-SIZE	2.4	5	A4	Z	14.4	288
635	2000	VOLVO	V70 GLT TURBO	STATION WAGON - MID-SIZE	2.4	5	A4	Z	13.6	274
636	2000	VOLVO	V70 T5 TURBO	STATION WAGON - MID-SIZE	2.3	5	A4	Z	13.9	274
637	2000	VOLVO	V70 T5 TURBO	STATION WAGON - MID-SIZE	2.3	5	M5	Z	13.0	260
638	2000	VOLVO	V70R AWD TURBO	STATION WAGON - MID-SIZE	2.4	5	A5	Z	14.7	299

We can see that there are four numerical pieces of data in our DataFrame. Those are `ENGINE SIZE`, the size of the engine in litres, `CYLINDERS`, the number of cylinders in that engine, `FUEL CONSUMPTION`, the consumption of fuel in kilometres per litre (interestingly in europe this is normally given by litres per kilometre), and `COEMISSIONS`, the CO_2 equivalent emissions in grams per kilometre. Note that all of these cars are from the year 2000, so their fuel economy is not fantastic and their CO_2 emissions are significantly higher than today's cars.

Now we implement the linear regression model.



```python
# Specify the features and the target.
X = fuel[['ENGINE SIZE', 'CYLINDERS', 'FUEL CONSUMPTION']]
y = fuel['COEMISSIONS ']

# Split the training and test data.
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2, random_state = 42)

# Initialise the regressor.
emission_model = LinearRegression()

# Fit the model.
emission_model.fit(X_train, y_train)

# Score the model.
emission_model.score(X_test, y_test)
```

```
0.9846361749926678
```

This time, rather than accuracy, the model score we receive is called the **coefficient of determination**, or R^2. This number is given by

$$R^2 = 1 - \frac{SS_{\text{model}}}{SS_{\text{mean}}},$$

where SS_{model} is the sum of residual squares given by our model, and SS_{mean} is the sum of squares of residuals if we assumed (as per figure 5) that the best model would be to simply output the mean target value. You can also think of R^2 as the ratio of variance explained by the model to the total variance in the target. In this case, 98.4% of the variance in carbon emissions is explained by the engine size, the number of cylinders and the fuel consumption, which means our model performed incredibly well!

If we want to make predictions using our model, we can use a format like this:

```
# Make a prediction for a 4-cyclinder 1.6l engine with fuel consumption of 10.5
emission_model.predict(np.array([1.6, 4, 10.5]).reshape(1, -1))

>>> array([213.48269139])
```

The `.reshape` changes the array into a form that the model is expecting (normally there are multiple inputs to predict, so it would be 2-dimensional - we only provided one input so we change it to a 2-dimensional matrix). What we see is that given these inputs we expect to produce about 213 grams of CO_2 per kilometre. This is incredibly close to the actual value for a 2000 ACURA 1.6 EL with A4 transmission!

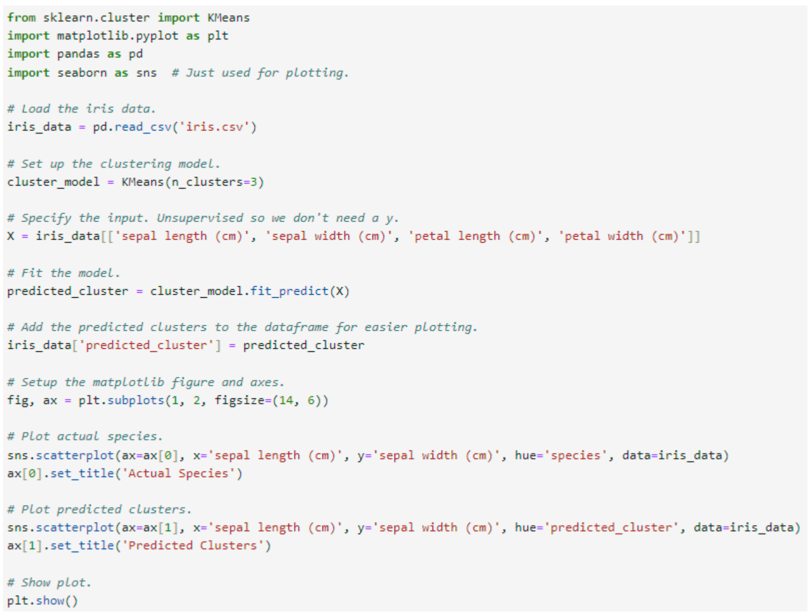
8.6 Some unsupervised techniques

While we aren't going to go too into depth into these two methods, it's worth mentioning them in case you want to use them in your own analyses.

8.6.1 Clustering

Suppose that you have a customer base for a company and want to find out if there are different **groups** of customers. Perhaps some are mid-day elderly shoppers, and others are 3am teenagers. Perhaps there's another group of working parents who only shop on Saturdays. These three groups are all likely to have different purchasing habits, and it might be useful to a company to be able to distinguish between people in these different groups. But without any prior knowledge, how could we detect where these groups are?

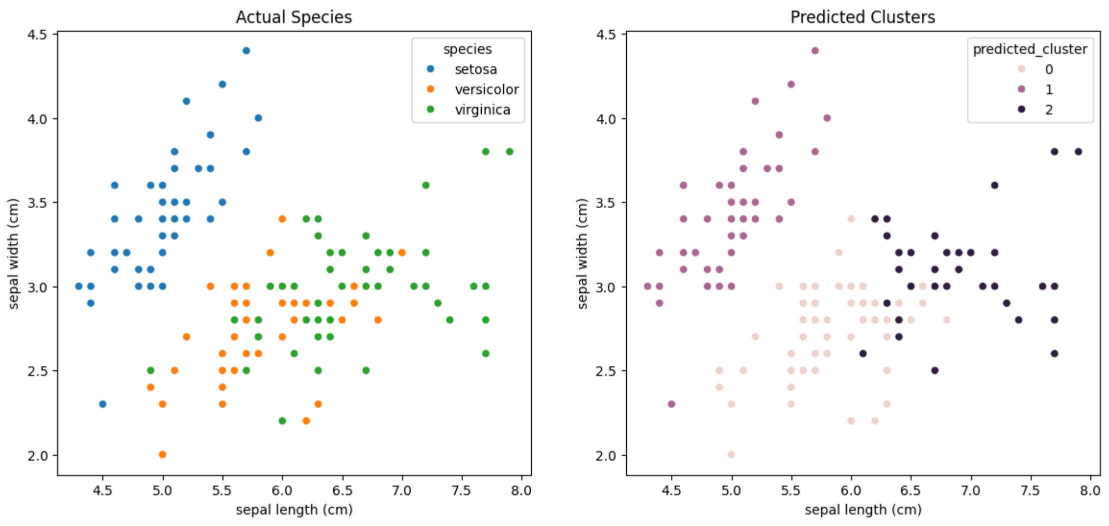
Methods which achieve this are called **clustering** methods, and scikit-learn also has several different implementations of clustering. Here, for example, is an implementation of clustering on the iris dataset again.



```
from sklearn.cluster import KMeans
import matplotlib.pyplot as plt
import pandas as pd
import seaborn as sns  # Just used for plotting.

# Load the iris data.
iris_data = pd.read_csv('iris.csv')

# Set up the clustering model.
cluster_model = KMeans(n_clusters=3)

# Specify the input. Unsupervised so we don't need a y.
X = iris_data[['sepal length (cm)', 'sepal width (cm)', 'petal length (cm)', 'petal width (cm)']]

# Fit the model.
predicted_cluster = cluster_model.fit_predict(X)

# Add the predicted clusters to the dataframe for easier plotting.
iris_data['predicted_cluster'] = predicted_cluster

# Setup the matplotlib figure and axes.
fig, ax = plt.subplots(1, 2, figsize=(14, 6))

# Plot actual species.
sns.scatterplot(ax=ax[0], x='sepal length (cm)', y='sepal width (cm)', hue='species', data=iris_data)
ax[0].set_title('Actual Species')

# Plot predicted clusters.
sns.scatterplot(ax=ax[1], x='sepal length (cm)', y='sepal width (cm)', hue='predicted_cluster', data=iris_data)
ax[1].set_title('Predicted Clusters')

# Show plot.
plt.show()
```

Note that `fit_predict` performs both the fit and predict methods simultaneously. It's available for most estimators in scikit-learn, to help keep your code concise. You can see that `KMeans` actually does a decent job of this clustering on the iris dataset.

While some of the members of different species cannot be differentiated by the clustering algorithm, broadly speaking, our unsupervised method was able to roughly characterise the three species, without having any prior knowledge. Clustering methods are incredibly powerful, so you might find them useful from time to time.

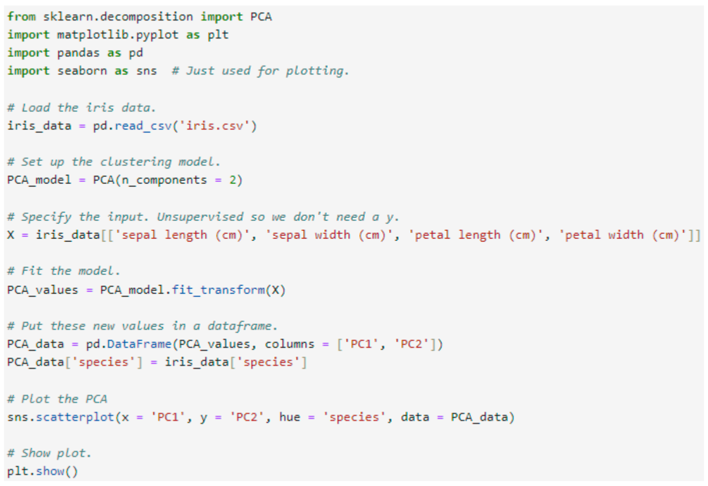


8.6.2 Principal Component Analysis

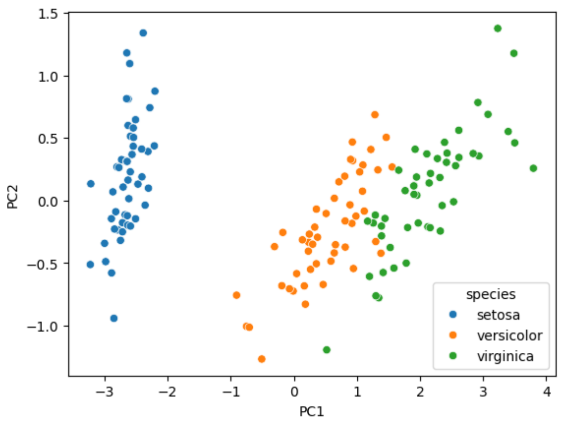
Suppose that you have a lot of different features that you're struggling to make sense of, and would like to reduce the number of degrees of freedom in your data. For example, if you see that two of your features always seem to increase together in near-perfect correlation, you don't necessarily need both of these features to get the same information.

This is where **dimensionality reduction** comes in. The most common dimensionality reduction is called **principal component analysis**, and it breaks down large numbers of features into a smaller number of features (called **components**), which capture as much of the variance in the data as possible.

Often this is useful to see where the real variance in your data lies, and plotting your new features, the principal components, can provide you with maximum spread in your data with as few dimensions as possible. We won't go too into depth here, but here's a quick implementation for the iris dataset again.


```python
from sklearn.decomposition import PCA
import matplotlib.pyplot as plt
import pandas as pd
import seaborn as sns  # Just used for plotting.

# Load the iris data.
iris_data = pd.read_csv('iris.csv')

# Set up the clustering model.
PCA_model = PCA(n_components = 2)

# Specify the input. Unsupervised so we don't need a y.
X = iris_data[['sepal length (cm)', 'sepal width (cm)', 'petal length (cm)', 'petal width (cm)']]

# Fit the model.
PCA_values = PCA_model.fit_transform(X)

# Put these new values in a dataframe.
PCA_data = pd.DataFrame(PCA_values, columns = ['PC1', 'PC2'])
PCA_data['species'] = iris_data['species']

# Plot the PCA
sns.scatterplot(x = 'PC1', y = 'PC2', hue = 'species', data = PCA_data)

# Show plot.
plt.show()
```

Note that as we are not explicitly making any predictions, the equivalent function to `fit_predict` in an unsupervised model is `fit_transform` in scikit-learn. Here we do this to extract a new representation of the data and save it to a new DataFrame.



Plotting these two principal components in the data help us to maximise the spread between these different groups using only 2 dimensions. Contrast this with the earlier plots where the data seems to overlap quite considerably. Clever use of PCA can be incredibly helpful in all sorts of disciplines, as PCA helps to find simplified representations of complex data.

8.7 Large language models

It would be remiss of me in 2024 to teach a course on applied mathematics and machine learning which made no explicit mention of large language models (LLMs) - the massively pervasive machine learning phenomenon of the 2020s. Large language models are supervised prediction models, whose structure is based upon a neural network architecture called a **transformer** [13]. While it is possible to implement neural networks with scikit-learn, we didn't discuss them in this course as they are often somewhat inefficient for relatively simple prediction tasks, and it's important to highlight that machine learning is more than just neural networks.

For complex, high-order tasks like language cognition, large artificial neural networks are incredibly powerful. The depth of their representation is only bounded by the depth of the network, which, for many of todays systems, is very large indeed. For example, GPT-4 is rumoured to contain 1.76 trillion parameters [1], vastly outnumbering the 100 billion or so neurons in a human brain [10]. If you haven't encountered an LLM yet, I highly advise you to try GPT-3.5 at `chat.openai.com`.

Large language models tend to have accuracy in their responses which scale with their architecture and compute (i.e. how much computation is used to train them). Despite this, even the best models like GPT-4 and Claude 3 can and do still confabulate - that is, they make things up occasionally, with exceptional confidence.

These models work by taking a large amount of text, represented as a list of tokens (small collections of symbols) as input. The input to the model is called the **context**, and the model will respond by predicting the next token in the sequence. Repeating this cycle, the model can generate long strings of text with human-like readability.

Because they're so powerful, it's worth briefly discussing how you can use language models to supplement your learning (be that now or at university), without falling for their occasionally very convincing confabulations.

If you want to see one of the most confusing responses from a language model that completely went off the rails, see this link (warning: contains swearing, religious themes, discussion of paralysis/illness, death, and otherwise unsettling material)
`https://chat.openai.com/share/f5341665-7f08-4fca-9639-04201363506e`.

8.7.1 Learning from language models

The best way to view a language model is a colloquial and friendly assistant who can give you pretty general overviews on a topic, but whose facts sometimes need correcting. This can be incredibly helpful sometimes if you're looking for ideas on how to start a project or what might be interesting to explore.

Language models are not currently very good at mathematical reasoning, so if you're going to use an LLM to help you understand mathematical concepts, try and ask it broad questions about the context. "Why should I learn about linear models?" or "How could learning about dimensionality reduction help me think about economics?" would be sensible and pretty safe questions to ask the machine.

In contrast, asking "Please solve this quartic equation: $x^4 + 2x^3 - x^2 + 2 = 0$." is unlikely to yield any fruitful results. It would likely simply give you vague ideas about possible approaches. Similarly, "Please prove theorem x" is unlikely to give you a particularly strong mathematical argument - even though it might look incredibly convincing.

In addition to broad contextual answers, LLMs do seem to be absolutely phenomenal for writing code. Indeed, many times while writing this course I've had the assistance of GPT-4 for quickly finding sensible solutions to some data manipulation problems. Do this within reason, however. It's often best to provide some of the code to the model yourself and ask for small changes, rather than have it construct an entire program that you will, inevitably, fail to completely understand.

Obviously refrain from plagiarising from LLMs. Generally they're not trained to provide references (although some like Perplexity's system built on GPT-4 can be helpful), and if you ask the machine to do your work for you, you're not going to get the in-depth understanding of the topic you expect. Consider sat-navs for example. While a sat-nav can tell you where to go and often get you to the right location, using a sat-nav actually appears to decrease your understanding of your environment, as you're not forced to actively engage with information about your location any more[4].

Despite this, it's becoming increasingly likely that healthy use of LLMs will start form a useful tool in your education which previous generations have not had access to. Learning to use them correctly can massively enrich your learning. Using them poorly will do the opposite, so be careful!

8.8 Good statistics, bad statistics

As a last final note in this course, I would like to remind you how important it is to use statistics carefully and with good basis. Poor statistics can mislead and hinder the progress of science, so it's incredibly important to make sure that, whatever you find, you check that your findings are actually real.

Consider the following. If every paper published performs 1 statistical test at the 5% significance level, then it could be that as many as 1 in every 20 papers contains a result **which doesn't even exist**. That doesn't make science sound as compelling as it should be.

To that end, here are some important things to keep in mind when you're using statistics and machine learning models in your work.

Adjust your p-values for multiple tests. If you do lots of tests for a single project, make sure that you don't stumble across false results by mistake.

Set your hypotheses before you perform your tests. If you look through your data for patterns and then perform tests, this is called *data dredging* and it usually finds spurious relationships.

Run your tests at a pre-determined p-value. Don't perform the test first, look at the p-value and then decide what counts as a significant result.

Report null results where possible. Science often suffers from a *publication bias*, where insignificant results are not published, even though they're valuable scientific information. One way to combat this is via *preregistration*, where you post ahead of time what tests you plan to perform. Some journals will accept some preregistered papers which later found null results to combat the publication bias.

Use good sample sizes. Small samples often skew results so that patterns more easily emerge. Good analysis should account for the expected skew in small samples.

Don't manipulate your collection methods. Don't stop your data collection early because you spot a pattern, or otherwise change your methodology in the middle of the experiment.

Be transparent about your data. Always endeavour to include the data you're analysing when releasing results. This allows other researchers to check your data is consistent and makes repeatability much easier.

When using ML models, NEVER TEST ON YOUR TRAINING DATA. This is a big one. Testing your models on your training data is such a big no-no, and will almost **always** show you better results than on newly sampled data. This also applies to *information bleed*, where information in the training data accidentally provides information about the testing data.

Follow these rules and the broad philosophy of good statistics, and you'll help make science and discourse on important topics much easier and more transparent!

8.9 Adieu

Thank you so much for taking part in this course on some applied mathematical methods! I endeavoured to make it as useful as possible to everyone and to show you how creative you can be with computational work. Real-world data is complex and beautiful, and it can be inspiring to see old data in new ways.

Throughout this course we focused on developing some familiarity with python for the purposes of exploring your own projects later on. While we've only seen some of the stars of the data-science show - namely `NumPy`, `Pandas`, `matplotlib` and `scikit-learn` - these packages alone can already take you incredibly far, and they make working with data quite enjoyable.

I hope also that you've had a chance to see just how much fun working with data can be! if you have any feedback on the course or just want to ask some questions, feel free to come and ask me or contact me via email!

Keenan

Figure 6: Brought to you by the power of A Arthfual Intellliigence.

References

[1] Gpt-4 architecture, datasets, costs and more leaked. https://the-decoder.com/gpt-4-architecture-datasets-costs-and-more-leaked/. Accessed: 2024-04-23.

[2] Jupyter notebook at the python package index. https://pypi.org/project/notebook/. Accessed: 2024-03-25.

[3] Mars climate orbiter. https://science.nasa.gov/mission/mars-climate-orbiter/. Accessed: 2024-04-04.

[4] Satnav users risk losing their natural navigational skills, expert warns. https://www.theguardian.com/science/2016/mar/30/satnav-users-risk-losing-their-natural-navigational-skills-expert-warns#:~:text=People%20who%20rely%20on%20satnav,our%20innate%20way%2Dfinding%20abilities. Accessed: 2024-04-23.

[5] Why do physicists mention "five sigma" in their results? https://home.cern/resources/faqs/five-sigma#:~:text=In%20the%20case%20of%20the,a%20Higgs%2Dlike%20particle.%E2%80%9D. Accessed: 2024-04-03.

[6] Z-score and probability calculator. https://www.calculator.net/z-score-calculator.html?c2z=-1.11&c2p=&c2pg=&c2p0=&c2pin=&c2pout=&calctype=converter&x=Calculate#converter. Accessed: 2024-03-26.

[7] James Betker, Gabriel Goh, Li Jing, Tim Brooks, Jianfeng Wang, Linjie Li, Long Ouyang, Juntang Zhuang, Joyce Lee, Yufei Guo, et al. Improving image generation with better captions. *Computer Science. https://cdn. openai. com/papers/dall-e-3. pdf*, 2(3):8, 2023.

[8] Tim Brooks, Bill Peebles, Connor Holmes, Will DePue, Yufei Guo, Li Jing, David Schnurr, Joe Taylor, Troy Luhman, Eric Luhman, Clarence Ng, Ricky Wang, and Aditya Ramesh. Video generation models as world simulators. 2024.

[9] Mar Estarellas and Antoine Bellemare. Latent ecologies of the mind: Exploring harmonic synchrony and complexity in human brain signals and beyond. 2024.

[10] Anil Gulati. Understanding neurogenesis in the adult human brain, 2015.

[11] Fernando E Rosas, Pedro AM Mediano, Henrik J Jensen, Anil K Seth, Adam B Barrett, Robin L Carhart-Harris, and Daniel Bor. Reconciling emergences: An information-theoretic approach to identify causal emergence in multivariate data. *PLoS computational biology*, 16(12):e1008289, 2020.

[12] Madalina I Sas, Pedro AM Mediano, Fernando Rosas, Hillary Leone, Andrei Sas, Christopher Lockwood, Henrik J Jensen, and Daniel Bor. Synch. live: Collective problem-solving through flocking motion induces higher connectedness to others. 2024.

[13] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, □ukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[14] Bertie Vidgen and Taha Yasseri. P-values: misunderstood and misused. *Frontiers in Physics*, 4:6, 2016.