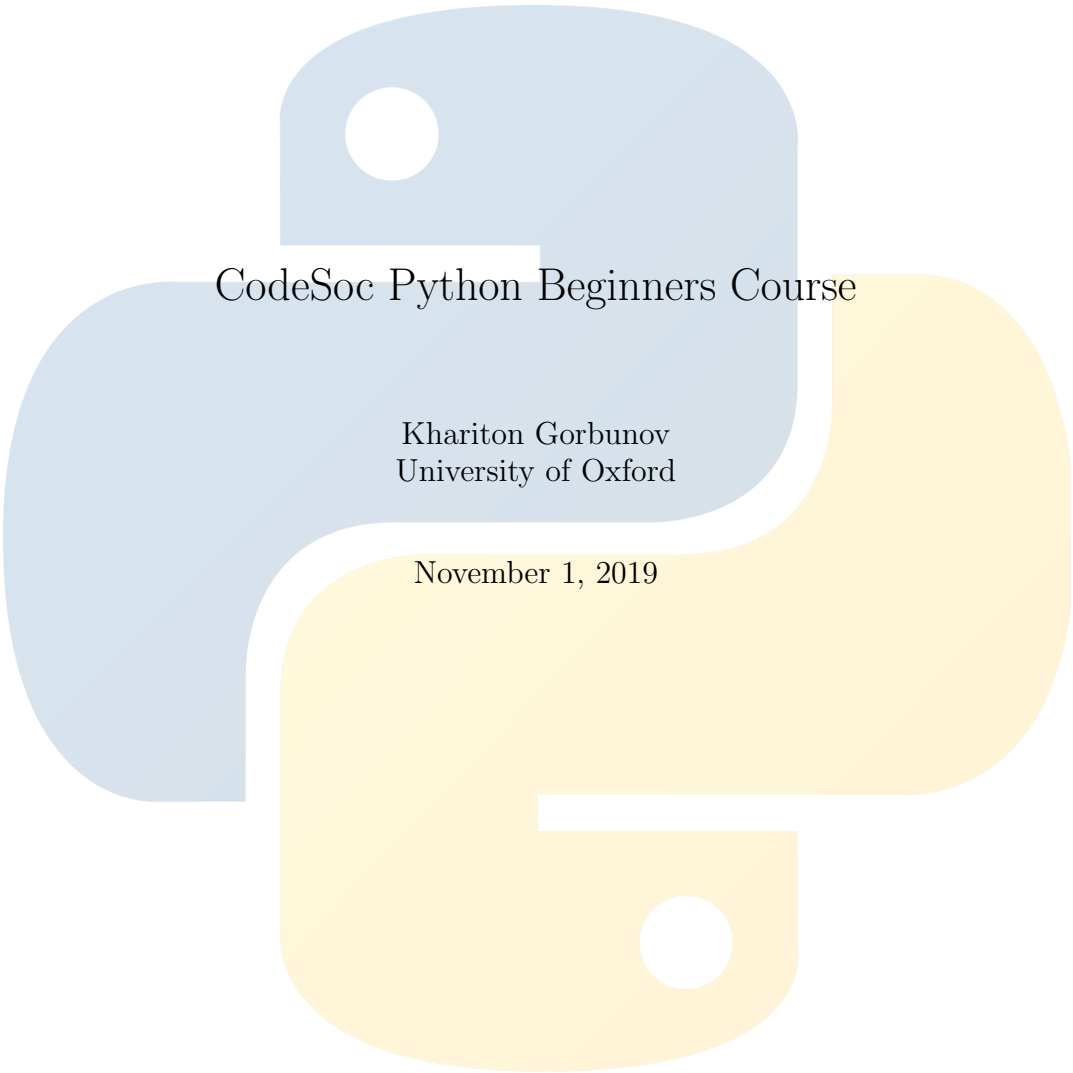

CodeSoc Python Beginners Course

Khariton Gorbunov
University of Oxford

November 1, 2019

Contents

1 Installation and Set Up

1.1 Overview

In this course we will use Python 3. That is the older brother of the widely used Python 2, which implements many new features. You may already have Python installed on your computer, so follow the installation instructions carefully to avoid any mistakes or double-installs. Python can take use of so called environments, which can be used to separate dependencies needed (which you will learn about later) for different projects, hence avoiding potential clashes. There are many ways to install Python, one of them being through a great manager called Anaconda, which comes with many pre-installed tools, and allows you to easily manage different versions and environments. For the sake of simplicity, in this course we will use native Python, which means directly installing it on to the machine.

To write Python code, we will need a text editor. We recommend Atom or Sublime 3, but any code-oriented text editor will do. Later in the course you will also learn how to edit and compile Python code in your web-browser with the help of a package called Jupyter Notebook.

1.2 Mac

To install Python 3 on a Mac, we will use a package manager called Homebrew. This requires XCode, a Mac OS development kit, so install it first by opening terminal (Command +Space and search "Terminal"). In terminal, run the following command:

```
$ xcode-select –install
```

When the installation is complete, install the Homebrew package manager:

```
$ /usr/bin/ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install)"
```

To install Python with Homebrew, run

```
$ brew install python3
```

To check whether Python 3 was successfully installed onto the machine, we can simply run a Python 3 shell by typing python3 into the terminal:

```
$ python3
```

The terminal should now be in Python mode, and to exit, simply type in "exit()" and press enter.

If all of this is successful, you should now have Python 3 installed on your machine.

1.3 Windows

Installing Python 3 on Windows requires you to download a Python installer, which can be found here: "https://www.python.org/downloads/windows/". Click on "Latest Python 3 Release - Python 3.x.x". After the installer has been downloaded, run it. Go through the installation, and it is recommended that the user leaves the installation directory as suggested by the installer. Make sure that "Add Python to PATH" is option is enabled, as this will give you access to Python via command prompt, saving time in the long run. After the installation is complete, open command prompt (search for CMD in the start menu), and type in:

```
python -v
```

The CMD should print the Python version installed. Make sure this is a Python 3 version and NOT Python 2. If it is Python 2, you will need to add Python 3 to PATH.

1.4 Writing and running your first Python program

If everything is installed correctly, you should now be able to run your very own Python program. It is a ritual for beginners to create the simplest of programs called Hello World. To do so, open your text editor and type in the following line:

```
print("Hello World")
```

Now save the file as HelloWorld.py (make sure to remember suffix .py for all Python scripts!) in a directory. It is suggested to keep an organised directory with all your scripts for this course, having a folder hierarchy similar to the structure of this course.

To run the script, open terminal for Mac or CMD for Windows. To execute a script, we need to tell Python where it is located. Rather than always copying the exact path to the file, we can simply enter a certain folder with our terminal. To do so, both Mac and Windows use the "cd" command (stands for "change directory"). So type in

```
$ cd your-directory
```

The code we just created was saved in documents/pythonForBeginners/Installation, so to set the terminal to that directory, we run:

```
$ cd documents/pythonForBeginners/Installation
```

To run the actual Python script, simply type in:

```
$ python3 HelloWorld.py
```

If everything went right, the terminal should now display "Hello World". If the terminal returns an error, it means that you either have errors in the code, incorrectly installed Python 3 or, mostl in case of Windows, Python 3 was not added to PATH.

2 Variables and data structures

Now that we know how to execute a Python script, we can start discussing how to actually create one. The script we created in the previous part simply writes out the phrase "Hello World" to the terminal using the "print" function. We will later discuss what functions are, but for now, simply accept that print will write out a variable to the terminal. So what are these variables? Do not let the name scare you, it is actually quite simple; A variable can be thought of as a container storing something. We can put something in that container, or we can simply view what is in it. There are different containers that store different types of stuff, for instance numbers or words.

The reason computer scientists make a distinction between these types of numbers is because of how these

numbers are represented in the binary system, and because each type requires different amount of memory, or in our metaphor, different variable types may require larger containers than others. This should intuitively make sense; larger numbers require more computer storage, equivalently to longer numbers needing more space on a piece of paper. In lower level languages, the programmer is required to specify the variable type, and in some cases, even allocate the memory manually. For instance, an integer takes up 4 bytes of memory. Luckily, Python is a high level language, which means the Python virtual machine allocates the memory for the user, or in simple terms, we do not need to worry about memory.

In this chapter we will look at different types of variables and their properties.

2.1 Numbers: Integers and floats

Most people reading this manual will have heard of integers. Plane and simple, integers are whole numbers, such as $-1, 0, 1, 2, ...$ Floats, however, have a rather special meaning in computer science, but as far as we are concerned, floats are types of variables representing the real numbers, such as $1.247, 1.0, -34.5, ...$ The difference is, as mentioned earlier, is how these numbers are represented, and how much memory they are allocated, but Python will recognise whether the variable is an integer or a float, and allocate memory accordingly. What the reader should keep in mind, is that the floats, or the real numbers will only have a finite length, so one has to take care when doing high-precision computation, such as numerical simulations or machine learning, as there will be a rounding error.

To define a variable in Python, simply make up a name, for example myVariable, and give it a numerical value using the "=" operator: $myVariable = 7$. This means that a variable (in this case of type Integer) has been created, given a value of 7 and stored in the memory. There are rules to the nomenclature accepted when naming variables, most of characters found on the keyboard can be used, for instance $a, b, c, 1, 2, 4, ...$, however some are reserved, usually most being operators, such as $*, +, -, =, !, >, <, ...$ Let us try to print our variable. Create a script called printVar.py, and in it write the following:

```
myVariable = 21
print(myVariable)
```

Run the script using

```
$ python3 printVar.py
```

and observe the output. The terminal should now print the value assigned to myVariable, 21. Python knows how to do mathematical operations on variables, so for instance, it can add, subtract, divide and multiply numbers. Create a script called addition.py, create a variable $a = 3$, $b = 2$, and let $c = a + b$, and display value of c to the terminal:

```
a = 3
b = 2
c = a+b
print(c)
```

Run the script, and the output should now be 5.

Now there are some things that should be kept in mind when writing code. "=" is not a mathematical equality. It is called the assignment operater, which will take the value on the right hand side and store it in the container on the left. One can also overwrite the value of variables by using the assignment operator. For instance:

```
a = 3
b = 2
```

```
c = a+b
c = a*b
print(c)
```

will print 6 and not 5, as firstly, c was assigned a value of $5(2+3)$, but later changed to $6(2*3)$. For that reason, this is allowed:

```
a = 3
b = 2
c = a+b
c = 2*c
print(c)
```

and prints 10 to the console, as first, c is assigned a value of 5, and then it is assigned whatever value it currently stores multiplied by 2. Obviously, in mathematics, "c=2c" is an equation that leads to no solution $(1 = 2)$. Also, be wary that while multiplication of 2 numbers, a and b, can be written as ab, in Python, one has to use the multiplication operator, $a * b$, or there will be an error. Also, white spaces usually do not matter within a line, so it is equivalent to write c=2 and c = 2, it is simply a matter of preference. White spaces come into play later when we discuss code blocks. This will all come with practice. Now let us try and write a code that computes the area of a triangle and prints it to the terminal:

```
base = 5
height = 2
area=0.5*base*height
print(area)
```

2.2 Lists and strings

Python allows users to create something called lists, also referred to as arrays or vectors, but they basically represent a chain of variable containers that can store stuff. For example, we can create a list that stores 5 numbers. This is not only more convenient than creating 5 separate variables, but also allows for a systematic nomenclature. Elements of lists can be accessed through indexing, as illustrated in figure 1. Indexing begins

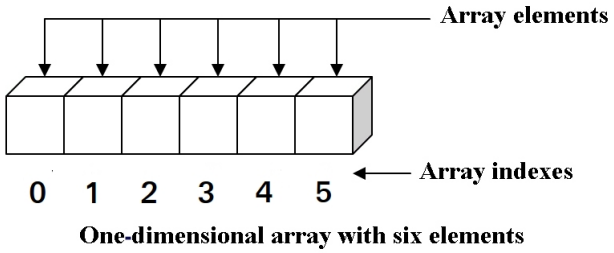

Figure 1: Array structure

from 0 and goes to length-1, so in the case of figure 1, the elements are numbered $0, 1, 2, 3, 4, 5$, but the length of that list is 6. Let us now try and create a list:

```
myList=[25, 7, 3.14, 9, 100, 1.2]
```

A list is enclosed with square brackets [], and the elements are separated with commas. So our 0th element is 25, 1st is 7, 2nd is 3.14, 3rd is 9, 4th is 100 and 5th is 1.2. We can also access certain elements of the

list. Say we wish to acces the 2nd element of our list. We do that using the square brackets ON the list as following:

```
myList=[25, 7, 3.14, 9, 100, 1.2]
print(myList[2])
```

The terminal will print 3.14. We can also index the list using a variable, but remember that the variable has to an integer:

```
myList=[25, 7, 3.14, 9, 100, 1.2]
i=3
print(myList[i])
```

This, however, will lead to an error:

```
myList=[25, 7, 3.14, 9, 100, 1.2]
print(myList[2.718])
```

Also an error will be generated if the index exceeds length-1, so calling myList[6] will lead to an error. Another convenient thing that Python allows us to do is reverse-indexing. We can use negative integers to index the list from the end, so myList[-1] gives the last element, 1.2, myList[-2] gives the second to last element, 100, etc. Python also implements an inbuilt function (like print) that will tell us the size of our list. It is called len() and takes the list as an argument inside the parenthesis. So for example:

```
myList=[25, 7, 3.14, 9, 100, 1.2]
print(len(myList))
```

should print 6. Also, for a more cleaner look we can assign the length to a variable:

```
myList=[25, 7, 3.14, 9, 100, 1.2]
lengthOfList=len(myList)
print(lengthOfList)
```

and produces the same output. Such use of intermediate variables is good coding practice, especially when the user intends to use that variable several times.

Let us now move on to discussing strings. Strings are basically variables that contain words and phrases. They are assigned using single (') or double ("") quotation marks. We can create a string the same way we created an integer:

```
country="Norway"
print(country)
```

The terminal prints Norway. Strings can include anything on the keyboard, including most operators, but some things are special characters. So for instance if one wants to use quotation marks, then the string can be defined with single quotation marks, and the double quotation marks will then be displayed inside the string:

```
print('The name of my favourite language is "Python"')
```

displays The name of my favourite language is "Python". The reason we discuss strings after lists is that

strings can be thought of as lists containing single characters. One can index a string using the same notation:

```
phrase = "I love pasta"
print(phrase[2])
```

Will print the third character, which happens to be "l".

Both lists and strings can be concatinated, aka "glued" together using the addition "+" operator, so

```
list1 = [1,2,3]
list2 = [4,5,6]
string1 = "abc"
string2 = "def"
list=list1+list2
string=string1+string2
print(list)
print(string)
```

will output [1,2,3,4,5,6] and "abcdef". One can also index certain slices of a list using the square brackets and colon:

```
string = "abcdefgh"
print(string[2:7])
```

will print cdefg. Why? Because indexing will take everything from and including 2nd element until and EXCLUDING the 7th element. So in general, myList[n:m] will give all elements $n, n+1, ..., m-2, m-1$. Remember that indexing starts at 0. There are also many other functions Python provides to handle lists, and remember, in programming Google is your best friend.

2.3 Dictionaries

We will also briefly touch on a variable type called "dictionary" as it will be a very convenient structure for a lot of data related applications. A dictionary is similar to a list in the sense that it contains multiple elements, however the way values are referenced in dictionaries is different. Instead of using an index to access the value (myList[index]), values in a dictionary are accessed using a key, and in Python we refer to elements of a dictionary as key-value pairs, hence the name "dictionary". Suppose we wanted to make a little dictionary that translates the following words from English to Latin:

Hello	Salveo
Name	Voco
Country	Rus
Happy	Laetus

We wish that the user inputs the English word, and the program to return the corresponding word in Latin. The way we construct the dictionary is by separating key and value by colons, separating key-value pairs with commas, and putting everything between curly braces. If we wish to read a value from the dictionary, we use the same notation as in a list, except the index is substituted by the key as it appears in the dictionary. Example of the English to Latin dictionary:

```
engToLat={"Hello":"Salveo", "Name":"Voco", "Country":"Rus", "Happy":"Laetus"}
print(engToLat["Country"])
```

This prints "Rus" to the terminal.

We can also add new key-value pairs to existing dictionaries. This is done by using a function native to dictionary called "update". It is not very important to know how exactly it does what it does, but more so how to use it:

```
engToLat.update({"War":"Beloo"})
```

The dot means the function operates on engToLat, something we will see when we do functions and classes. Another thing to keep in mind about dictionaries is that keys should be unique, and in case of a redefinition, the last key-value pair will be the valid one.

2.4 Comments

It is now worth mentioning that the programmers can add comments to their code. Comments are lines that programmers add for themselves or other developers that may work on the code. These are lines that are not executed by the compiler, i.e. they are simply ignored when the program is run. To create a comment line, simply puts a hashtag "#", and then write whatever you like:

```
myVar=5 #set myVar to a value of 5
#now we wish to display the variable
print(myVar) #prints myVar to the console
```

3 Conditional statements and logical expressions

3.1 Basic logic

Conditional statements and logical expressions lie at the heart of programming and computer science. By logical statement, we are simply talking about truth tables of boolean variables. Boolean variables are such that they can either take on a value of "True" or "False", often reffered to as 1 or 0, on or off etc.

Let us work through an example of a truth statement. Let's say we have variables a, b and c that can either be true or false. Let us also construct a logical statement a or b = c. Intuitively, we know that if either a or b is true, then c is also true. It has the following truth table:

a	b	c
True	True	True
True	False	True
False	True	True
False	False	False

This is quite abstract, but logical expressions come into play in our lives every day, but they are usually quite simple and intuitive, and do not require truth tables (and the ones we will be doing in the course are too). Say for instance you are deciding whether to go to hall today or not, and you will only go if either the food or the drink they are serving is of your liking. Then the logical expression will be goToHall = likeFood or likeDrink, where goToHall, likeFood and likeDrink are boolean variables, while "or" is a logical operator.

Let us put this into code. You can assign a boolean value to a variable by typing in "True" or "False", and logical operators in Python are as in human language, so you can simply type "or", "and" etc:

```
likeDrink=True
likeFood=False
goToHall=likeFood or likeDrink
print("Going to hall?")
print(goToHall)
```

prints "Going to hall?" True to the terminal. Play around with these values and try to add the "and" operator.

Another important set of operators are comparison operators. Say we wish to know whether a value x is less than 10. We can then say $isLess = x < 10$, which means we assign the conditional "$x < 10$" (which is obviously either true or false) into "isLess". Try it in code.

3.2 If statement

Often, programmers wish to execute different pieces of code based on what previous results in the scripts. Say we wish to print "I am going to hall" only if goToHall is true. In that case we use something called an if statement. This is best illustrated using an example:

```
likeDrink=True
likeFood=False
goToHall=likeFood or likeDrink
if(goToHall):
    print("I am going to hall")
```

This will output "I am going to hall". Now try changing likeDrink to False, and running the program. It now outputs nothing. That is because the print statement is only executed if the argument of the if statement is true. Notice the indent before the print statement. That is to signify what is to be executed in case that the if statement is true. The general form is as following:

```
#executes when program is run
#executes when program is run
#executes when program is run
if(something):
    #executes only if "something" is true
    #executes only if "something" is true
    #executes only if "something" is true
    #executes only if "something" is true
#executes when program is run
#executes when program is run
```

In general, indents can be done with both tabs and spaces, but make sure to be consistent, otherwise the compiler will throw an error. Pieces of code with the same indent level are referred to as code blocks. We can make a nested if statement, which means we have an if statement in another if statement:

```
likeDrink=True
likeFood=False
goToHall=likeFood or likeDrink
if(goToHall):
    print("I am going to hall")
```

```
    if(likeDrink):
        print("The drink is going to be good")
    if(likeFood):
        print("The food is going to be good")
```

Running the program outputs "I am going to hall" "The drink is going to be good".

Also, keep in mind that goToHall here is an intermediary variable, and you could simply put "if(likeFood or likeDrink):", which intuitively makes sense. We can also use the operator "not" to print if we are not going to hall. "not" switches true into false and false into true:

```
likeDrink=True
likeFood=False
goToHall=likeFood or likeDrink
if(goToHall):
    print("I am going to hall")
    if(likeDrink):
        print("The drink is going to be good")
    if(likeFood):
        print("The food is going to be good")
if(not goToHall):
    print("I am not going to hall")
```

3.3 Else and else if

Imagine if there are a lot of parameters when it comes to hall food, for example likeFood, likeDrink and friendsPresent, but you only want to go to hall if the food is good (likeFood=True). The conditional to use then is an else statement. The else statement has to be placed after the if statement, and will be executed if that if statement is false:

```
likeFood=False
likeDrink=True
friendsPresent=True
if(likeFood):
    print("Going to hall")
else:
    print("Not going to hall")
```

This will print "Not going to hall". Suppose if you don't like the food, you will only maybe go to hall if your friends are there. Otherwise, you do not want to go. Then you can use an else if statement using keyword "elif" to describe that situation.

```
likeFood=False
likeDrink=True
friendsPresent=True
if(likeFood):
    print("Going to hall")
elif(friendsPresent):
    print("Maybe going to hall")
else:
    print("Not going to hall")
```

Will print out "Maybe going to hall". Try and combine different if statements and their counterparts and see how everything plays together.

4 Loops

The most boring tasks are often the repetitive ones, where the computation is the same for every step, just applied to a different piece of data. Examples of such are for instance calculating the variance of a dataset, generating personalised emails or simply writing out the first 100 even numbers. Python allows us to do such iterative actions through something we call loops.

4.1 For loop

Suppose we for some reason wanted to print out the first 10 integers. We could of course do "print(1), print(2), ...". But that is very lengthy and defies the purpose of programming. Also, what if this is a user defined range, and the user asks to print only the five first integers? Surely, you would not want to wrap all the ten prints in if statements? The for loop allows us to iterate through a range of values in a container (for example a list). Suppose we have a list numbers=[1,2,3,4,5], and we wish to get the square of each element. Pseudo-code for that scheme is:

1. define variable number=numbers[1]=1 (first element in our list)
2. define square=number**2
3. print(square)
4. if we have any more items in our list, let number be the next item in numbers and go back to step 2. If not, we are done.

The code for that is quite simple:

```
numbers=[1,2,3,4,5]
for number in numbers:
    square=number**2
    print(square)
```

prints "1", "4", "9", "16", "25" as expected. "number" is a variable that we defined, and by using keyword "in", we are sequentially taking values from numbers, assigning them to number, doing our operations(square and print), until there are no more items left. We can obviously use everything we learnt before inside the for loop. So suppose we wanted to print the squares, but only if the square number is greater than 15. We could do that by squaring the number, checking whether it is greater than 15, otherwise we want to print the original number:

```
numbers=[1,2,3,4,5]
for number in numbers:
    square=number**2
    if(square>15):
        print(square)
    else:
        print(number)
```

prints "1", "2", "3", "16", "25" as expected. Keep in mind that if you wanted to change the values of the original list, this method would not work, as number is a variable that takes on values in the list, but has no further connection to it. If you wanted to generate a list [1,2,3,16,25], you would want to reference the original list using the index notation. So numbers[i]=square if square greater than 15 for i=0,1,2,3,4. We could of course create a list index=[0,1,2,3,4] and use that:

```
numbers=[1,2,3,4,5]
index=[0,1,2,3,4]
for i in index:
    square=numbers[i]**2
    if(square>15):
        numbers[i]=square
print(numbers)
```

gives the desired numbers=[1,2,3,16,25]. However there is a more convenient way of generating these index lists, and that is with a Python function called "range()". Using range is quite straight forward; range(a,b) generates a list of integers from and a (inclusive) to b (exclusive), in other words $\{a, a+1, ..., b-1\}$ or $\{i \in Z | a \leq i < b\}$. So to get our list, we would say range(0,5), which would generate 0,1,2,3,4. So our code can be simplified:

```
numbers=[1,2,3,4,5]
for i in range(0,5):
    square=numbers[i]**2
    if(square>15):
        numbers[i]=square
print(numbers)
```

giving the same output.

Range generates all the integers in the range, but if we wish, we can add an optional which is common difference, so range(a,b,d) will give all numbers a, a+d, a+2d, ..., r, r¡b. So for instance if we only wanted to square every second element, we could do:

```
numbers=[1,2,3,4,5]
for i in range(0,5,2):
    square=numbers[i]**2
    numbers[i]=square
print(numbers)
```

returns [1, 2, 9, 4, 25]. There is a constraint on the common difference, and that it has to be an integer.

4.2 While loop

For loops are good when we know the dimension of the problem we wish to solve with the loop, such as length of the list or number of steps. However, this is not always the case. Say you want to compute the all Fibonacci numbers less than 200. A Fibonacci number is simply the sum of two previous Fibonacci numbers, aka $F_k = F_{k-1} + F_{k-2}$. We start with 0 and 1 as base numbers, so $F_0 = 0, F_1 = 1$, hence $F_2 = F_1 + F_0$, $F_2 = 0 + 1 = 1$. We can keep going, and we get something like 0,1,1,2,3,5,8,13 etc. However, doing this manually may get quite lengthy, so we can implement this using a loop. Let us do a pseudo-code, a powerful planning tool for programmers:

1. Define a current Fibonacci number, previous and the previous before that Fibonacci number, FCurr,

FPrev, FPrevPrev. Also define a list to store our numbers.

2. Set FPrevPrev=0 and FPrev=1, add them to out Fibonacci numbers list and calculate Fcurr.
3. Add the current Fibonacci number to the list, and shift all the Fibonacci numbers backwards, such that FPrev takes on value of FCurr and FPrevPrev takes on value of FPrev.
4. Calculate the new current Fibonacci number, and if it is less than 200, repeat from step 3.
5. Print out the list of numbers

And here is the code using a while loop:

```
#intitialize the variables
FPrevPrev=0
FPrev=1
FArr=[]
FCurr=FPrev+FPrevPrev
#add them to the list
FArr.append(FPrevPrev)
FArr.append(FPrev)
#repeat while FCurr is less than 200
while FCurr<200:
    #append the current Fibonacci number
    FArr.append(FCurr)
    #shift the variables
    FPrevPrev=FPrev
    FPrev=FCurr
    #Calculate the new Fibonacci Number
    FCurr=FPrev+FPrevPrev
print(FArr)
```

The output is [0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144]. As you can see, every iteration, the while loops checks whether the condition is true, and if it is, then it keeps repeating. If it is false, the loop is complete. While loops are very useful, but can also be quite dangerous. There may be instances where the problem becomes very large without the programmer being aware of it, and may take very long to compute. Also, one may accidentally enter an infinite loop, which means the loop never terminates. For instance imagine if we forgot to update FCurr in the loop. The loop would then run forever, and may even lead to the computer crashing!

4.3 Break statement

Sometimes a programmer may wish to terminate a loop prematurely. Say you are looking for a particular item in a list; if you find it, you do not want to keep looking. Then a keyword called "break" can be used. If we have a list of students, and we wish to know whether Jane is in it, we can write a loop to go through the list. If Jane is an element, we break the loop. That means the loop just ends, even though the specified condition is not necessarily reached:

```
names=["George", "James", "Jane", "Maria", "Chris"]
found=False
for name in names:
    if(name == "Jane"): #== is the comparison operator
        found=True
        break
print(found)
```

When the loop hits "Jane", it no stops executing, so it never checks the names "Maria" and "Chris". In this case, it simply reduces computation time, but break statements can have functional uses as well.

5 Functions

5.1 Basics

Many people will be familiar with the concept of functions from mathematics, and some may even find them dreadful. Worry not. A function in programming draws parallels to the ones in math, but ultimately can be thought of machines that take an input and spit out an output. Functions are simply recipes, and that recipe is defined by the user. Suppose you are working on a program with geometrical applications, and

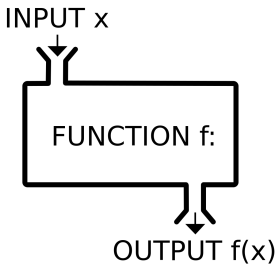

Figure 2: Function as a machine

many times throughout, you wish to compute the area of a circle based on its radius. Yes, you could always just write out the equation for area of a circle every time, but what if the calculation is longer? Such as area of a polygon with n vertices? That could become lengthy and quite messy, so it would be much better if we could just say something like "get area" and receive the area of the circle. This is what a function does. Mathematically, the area function can be written as $A(r) = \pi r^2$. So when we ask of this function A(5), we get back 78.5.

To create a function in Python, we use the keyword "def" followed by a functions name and its arguments in parenthesis, so for our area function we can do something like:

```
def getArea(radius):
    #recipe goes here
```

Inside that code block, the recipe is written. The variables created inside the function will not spill out on the rest of the code. Functions are meant to be abstract, and it is good coding practice to design functions to be self contained, in other words, it should not reference any variables outside the function. For our area function, we wish to call "getArea" specifying a radius, and get an area back. To do so, we can fill in the recipe, and return the value to the user:

```
def getArea(radius):
    area = 3.141*radius**2
    return area
```

The "return" keyword simply spits out the desired output. So we can do something like this:

```
def getArea(radius):
```

```
    area = 3.141*radius**2
    return area

myArea = getArea(5)
print(myArea)
```

The output is 78.5 as desired. When the script is executed, the compiler will simply scan the function and memorise that it exists, but no function will be executed unless it is explicitly called. Something to keep in mind is that a function cannot be called before it is defined, and will lead to an error.

5.2 More on arguments and return

A function does not need to do something mathsy. In fact a function can be anything! We can create a function that compares the age of two people, and prints the output to the terminal. The creator of the function can define the input type, so lets say we define people as an array where the first element is the name, and the second is age, something like person=[”Jon”, 35]. Then we wish to pass two people into the function and print their names:

```
def compareAge(person1,person2):
    age1=person1[1]
    age2=person2[1]

    name1=person1[0]
    name2=person2[0]

    if(age1>age2):
        print(name1+" is older than "+name2)
    elif(age1<age2):
        print(name2+" is older than "+name1)
    else:
        print(name1+" and "+name2+" are the same age")

alex=["Alexander", 22]
jamie=["James", 42]

compareAge(alex, jamie)
```

Outputs ”James is older than Alexander” to the terminal.

The function takes in two arguments, separated by a comma. When you use a function, make sure to pass it the variables in the order in which they are specified in the function. Also, look how the function is an abstraction; it does not matter what variable you pass it, as long as it is in the right format, the function simply compares an arbitrary person1 and person2, and in our case, person1 takes on the value of ”alex” and person2 takes on ”jamie”.

You may also notice that there was no return statement in the function. A function does not have to return something. Here, we simply stated the function, and since it had print statements in it, that is all we needed to do. In fact, a function does not even have to have an argument! We can even do something silly as this:

```
def bark():
    print("woff")
bark()
```

will print "woff" to the terminal.

That being said, returns can e very useful. Once the program hits a return statement, it immediately terminates the function, so anything after a return statement is not executed. This can be used similarly to a break statement; let's say we have an ordered list of names, and we wish to find the first person whose name starts with a particular letter. In that case, we can write a search function:

```
def getNameFromLetter(names, letter):
    for name in names:
        if(name[0]==letter):
            return name
    return None

class2019=["George", "James", "Jane", "Maria", "Chris"]
nameFound=getNameFromLetter(class2019, "J")
print(nameFound)
```

will print "James", as it is the first name that appears. You see that "None" is not printed, as once the function hits the return statement, it terminates. However if we use "K" for the letter, the print will be "None".

5.3 Recursion

Remember the Fibonacci numbers we calculated earlier? To do so, we had to create a loop, and keep track of how many Fibonacci numbers we had calculated. There is a more elegant way to solve this. The definition $F_k = F_{k-1} + F_{k-2}$ is a recursive statement, meaning that the next value depends on the preceding values. We can therefore write a function that calls itself, also known as a recursive function. We have to be careful though; the function cannot just keep calling itself, there has to be something we call base cases. Base cases are non-recursive return types. In our case, it is sensible to let base case be Fib(1) and Fib(0). We know that the 0th Fibonacci number is 0, and the 1st one is 1, so we can simply return those when n hits 1 or 0.

```
def Fib(n):
    #base case 1:
    if(n==1):
        return 1
    #base case 0:
    elif(n==0):
        return 0
    #recursion
    return Fib(n-1)+Fib(n-2)

print(Fib(4))
```

Prints 3 as expected.

The function is being called with an argument of 4, it checks the base cases, and since they are not true, it calls itself with 4-1 and 4-2, outputs of which we wish to add. Recursive functions can be quite confusing, so it is helpful to visualise the call stack. The call starts at Fib(4), and is pending. The calls go down the branches until base cases are hit, which are Fib(1)=1 and Fib(0)=0. When that happens, the returned values are sent up the branches, pending calls are executed (in reverse order of the calls), all the way up until Fib(4).

One has to be wary with recursive functions. If you take a look at the call stack you will see that Fib(2)

and Fib(0) is computed twice, and Fib(1) is computed 3 times. This is inherently inefficient, and you can just imagine the callstack of Fib(12). Fib(11) and Fib(10) will compute almost the same values, as Fib(11) requires Fib(10) and Fib(9). Try and execute the function for Fib(30). How about Fib(32)? Do you see a delay? Depending on the power of your computer it may or may not be noticable, but trying to compute the 100th Fibonacci number with this function would be madness, and one should rather use the for loop approach, or even pen and paper. A way to fix this issue is to use dynamic programming, aka store some of the values to remove the double computations, but this is beyond the scope of this course.

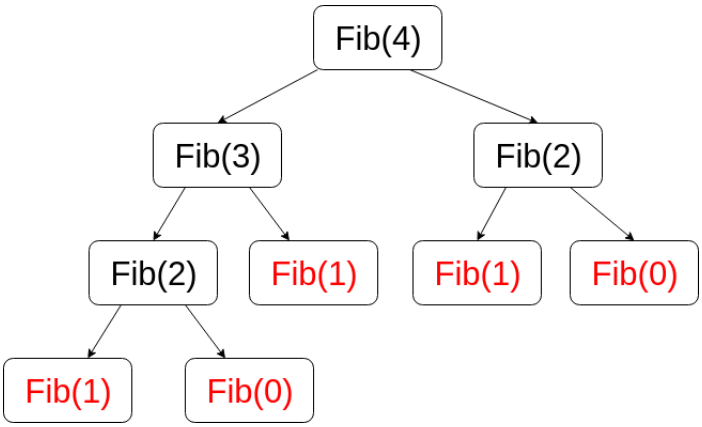

Figure 3: Fibonacci function callstack

6 Classes

Until now, we have dealt with arrays to store values we might want to access. We did examples where we stored peoples names and ages as a list of lists, something like: people=[[”Joe”,21], [”Yoshi”, 16], [”Rick”, 21]], where the first element corresponds to the first person, so person1=people[0], and the corresponding name is the first element of that again, so name1=people[0][0]. This approach works for certain complexities, but is rather unstructured and messy.

Suppose we somehow wanted to define a structure, which stores parameters of a person, such as name, age, address etc. We do that by defining something called a Class. A class is an abstract structure like functions, except a function is a blueprint for an executable (something we simply call and it does things according to a recipe), whilst class is blueprint for an object.

All data types we have dealt with such as integers, floats, lists etc can be considered objects. There is an underlying class that defines the common behavior of all these objects stored in the Python library on your computer. To illustrate that variables are different types of objects, we can call the ”type()” function on different variables, and observe the output:

```
var1 = 2
var2 = 3.141
var3 = [1,2]
var4 = {"key1":"val1"}
var5 = "hello"

print( type(var1) )
print( type(var2) )
print( type(var3) )
print( type(var4) )
print( type(var5) )
```

Prints class 'int', class 'float', class 'list', class 'dict', class 'str'.

This shows that every variable type is distinct from one another, and follows different rules based on the underlying blueprint, ie the class.

If we wanted to implement a class that defines the structure of people, we simply use the keyword class, followed by the name:

```
class Person:
    #blueprint goes here
```

To create an instance of that object, we simply assign it to a variable:

```
class Person:
    pass
    #blueprint goes here

person1=Person()

print( type(person1) )
```

Do not worry about the "main" part, the key takeaway is that the variable "person1" is now of type "Person". Also, the keyword "pass" basically means do nothing, and we put it there because we haven't put in anything else in the class yet.

6.1 Internal variables

If we wanted to define name and age as parameters of Person, we can just put then in directly into the class:

```
class Person:
    name=None
    age=None

person1=Person()
person1.name="Joe"
person1.age=30

print(person1.name)
print(person1.age)
```

Prints Joe, 30.

First of all, we defined name and age inside the Person class and initialised them to None. This is because we do not know the names of the people for whom we may want to instantiate the class. Next, in the actual code, we defined a person, person1, and set the name and age to "Joe" and 30. We did that through the "." dot operator, which allows us to access internals of the class. This is similar to append function used with lists, where we would say list.append(item).

We can consequently define member functions:

```
class Person:
    def greet(self):
        print("Hello!")
```

```
    name=None
    age=None

person1=Person()
person1.name="Joe"
person1.age=30
person1.greet()
```

Prints "Hello!"

We call greet with the dot operator as it is a member function. The odd thing here is that we call greet with no arguments, but the function blueprint takes in argument "self". This is because every function in a class wishes to take in a subject, and the keyword self passes the function the instance from which it is being called.

Now the advantage is that we can create multiple instances of the same class:

```
class Person:
    name=None
    age=None

person1=Person()
person1.name="Joe"
person1.age=30

person2=Person()
person2.name="Anna"
person2.age=50

print(person1.name)
print(person1.age)
print(person2.name)
print(person2.age)
```

Prints Joe, 30, Anna, 50

This displays how class Person is not a variable but a blueprint, of which we can create different instances with their own, unique values assigned to the parameters.

6.2 Constructor

In the previous subsection we used the dot operator to access and assign values to instances. However, the nature of our problem may require certain parameters to be defined on instantiation, similar to a person having to be named at birth. Therefore we use something called constructor. It is the proper method through which we create an instance and define certain internal variables. This is better illustrated through an example:

```
class Person:
    def __init__(self, name):
        self.name=name

person1=Person("Joe")
print(person1.name)
```

Here we see that when the object is first instantiated, we give it a mandatory argument name. What happens under the hood is that we have a special function, __init__, which is called upon instantiation, and is therefore called the constructor. Special functions in Python are often wrapped in double underscores, and have special meanings. Inside the constructor, we define a instance variable "name", and assign to it the value from the arguments of the constructor. Good coding practice is to give arguments different name than the actual instance variable, so something like this:

```
class Person:
    def __init__(self, givenName):
        self.name=givenName

person1=Person("Joe")
print(person1.name)
```

There is so much more to classes than this, such as classes being subclasses of other classes (inheritance), polymorphism and privacy settings, but that is beyond the scope of this course. The most important point of this section is to make the reader familiar with abstract structures and the syntax, as many of the external libraries used will implement their own objects with certain functionalities, and it is important to understand the syntax and how to leverage these properties.

7 Imports and external libraries

7.1 Local imports

So far we have written classes, functions and their applications all in the same file. However, that can get very messy and makes the code less recyclable. It is possible to separate out functions and classes you may want to use in the future into a separate file. For this demonstration, do not use jupyter notebook but use native Python and the File Navigator.

Suppose we are doing data analysis, and we wrote functions to calculate mean and variance of a dataset, and we use them on a particular dataset:

```
def mean(data):
    mean=0
    for i in data:
        mean+=i
    return mean/len(data)

def variance(data):
    variance=0
    ave=mean(data)

    for i in data:
        variance+=(ave-i)**2
    return variance/len(data)

prices=[1,2,3]
print("the average price is ", mean(prices), " and the volatility is ", variance(prices) )
```

You can imagine how these are functions you may want to reuse for future scripts, avoiding the tedeousness and messiness of copy-pasting them into the new script. Therefore, we can put them into a separate file.

In a given directory, create a file called **stats.py** containing the following code:

```
def mean(data):
    mean=0
    for i in data:
        mean+=i
    return mean/len(data)

def variance(data):
    variance=0
    ave=mean(data)

    for i in data:
        variance+=(ave-i)**2
    return variance/len(data)
```

This will now be a statistics library you can reuse. To do so, create a working script (the project you may be working on) and save it **in the same directory**. Call it **workingscript.py**. To reuse the library in the new script, use keyword "import" followed by the name of the file from which we are importing assets **excluding** the .py part:

```
import stats
```

We have now essentially told the computer to consider a module called stats in the working script. To use the modules, we need to reference stats using the dot operator:

```
import stats

prices=[1,2,3]
print("the average price is ", stats.mean(prices), " and the volatility is ",
    stats.variance(prices) )
```

To execute the script, in terminal we again type in:

```
$ python3 workingscript.py
```

Note that we do not need to call any of the libraries from terminal, only the working script. If a module has a long name, we can also use the keyword "as" and define a name by which we are referencing the module:

```
import stats as st

prices=[1,2,3]
print("the average price is ", st.mean(prices), " and the volatility is ", st.variance(prices) )
```

7.2 External libraries

It may upset you to find out that most of the method and algorithms, such as Fibonacci calculator, mean and variance functions already exist, and you can easily access them by downloading and importing external libraries. Similarly to importing homemade libraries, one can download and import modules from the internet and freely use them in their code. Certain modules come pre-installed, such as for instance the math module:

```
import math

print(math.pi)
print(math.factorial(5))
print(math.fsum([1,1,1]))
```

These particular functions and variables included in math are quite intuitive in terms of input/output. However, this may not always be the case. Also, there is no way of knowing how a module works, and most of the times, that is fine. Programmers usually treat external modules as black boxes, only caring about input/output types. Certain functions may have constraints, and it is best to find out specifics by reading the module documentation.

The math module is very well documented on docs.python.org, whist some modules have messy and incomplete documentations. Such is life.

While some modules come pre-installed, which we can just access through "import" keyword, most modules you may want to use require download and installation. Python provides the users with a repository and a package manager called pip. To install a module X, type in your terminal:
`$ pip3 install X`

7.3 Numpy

Numpy is a module that provides users with tools needed in numerical computation. To install numpy, run
`$ pip3 install numpy`

Now we can write some code:

```
import numpy as np

a=[1,2,3,4]
a=np.array(a)
print(a)
```

Prints [1 2 3 4]

Numpy has its own list-equivalent data structure called numpy array. It implements usefull functions for numerical computation. For instance, we can do matrix multiplication $A\boldsymbol{b}$:

```
import numpy as np

b=[[1],[2]]
A=np.array([[2,3],[4,5]])
c=np.matmul(A,b)
print(c)
```

Sometimes, modules will have sub-modules, so for instance numpy has a linear algebra submodule called linalg. This module is for instance required to calculate inverse. To implement the following computation: $A\mathbf{x} = \mathbf{b}$ solving for $\mathbf{x}$, $\mathbf{x} = \mathbf{A^{-1}b}$, can be done with the following code:

```
import numpy as np
```

```
b=[[1],[2]]
A=np.array([[2,3],[4,5]])
Ainv=np.linalg.inv(A)
x=np.matmul(Ainv,b)
print(x)
```

prints [[0.5], [0]] as expected.

Note how the inv() function is referenced through both numpy and its submodule linalg. Sometimes, we do not wish to import a whole library. We can then target modules to import using the "from" keyword:

```
from numpy.linalg import inv

Ainv=inv([[2,3],[4,5]])
print(Ainv)
```

Prints the inverse of the matrix, without us requiring to import all modules of numpy. This can be really useful in terms of optimisation.

There are many more applications of numpy, mostly dealing with tensors, linear algebra and numerical computation. It should be noted that some modules are meant to be used with numpy, such as tensorflow, which is a machine learning module.

7.4 CSV

Often, we are dealing with externally saved files such as .csv, text files and images. We will now go through an example of opening a .csv file. Place the csv file in the same folder as your script, or make sure to provide the full path in the code. Import the csv module, and open the csv file:

```
import csv

csvfile = open('GBP_USD_data.csv')
readCSV = csv.reader(csvfile, delimiter=',') #consult documentation for more info
for row in readCSV:
    print(row)
```

What we have done so far is to open the csv file, read it in and then print out all the rows. We have the raw data which we now can work with. Suppose we want to get prices and calculate some statistics. We can see from the print that the first row is not useful, and that in every row, we only need the second element. Also note that all data is now strings, so we need to convert the text price into an actual number. These things can all be summarised by the following code:

```
import csv
import statistics as stats

prices = []
csvfile = open('GBP_USD_data.csv')
readCSV = csv.reader(csvfile, delimiter=',')
for i,row in enumerate(readCSV):
    if i!=0:
        prices.append(float(row[1]))
```

```
avePrice=stats.mean(prices)
stDev=stats.stdev(prices)

print("Mean price is", avePrice, "and the standard deviation is", stDev)
```

prints out list of all the prices.

We initialised a list to store prices, and then we go through all rows except the first one and add the second element to prices, converting it from string to float. In addition, we imported the Python statistics module as stats, and used its functions to compute and display the mean and the standard deviation of the data set.

7.5 Matplotlib

For now we have been dealing with bunch of data, but at no point have we had the tools to visualise what we are doing. Therefore, it is time to introduce Matplotlib, a graphing package in Python. Install Matplotlib by running:

```
$ pip3 install matplotlib
```

After doing so, you can import Matplotlib into your code. We will be using a particular submodule called pyplot:

```
import matplotlib.pyplot as plt
```

So the first and easiest plot, which is of a single list:

```
import matplotlib.pyplot as plt

y=[0,1,2,3,4,5]
plt.plot(y)
```

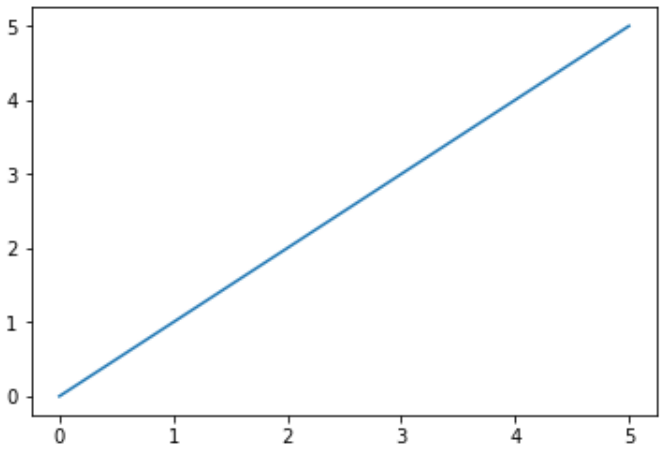

Figure 4: Simple plot

It is quite uncertain what has been plotted here. It looks like the identity function $y(x) = x$, $x \in [0, 5]$. Let us try some different values to figure out what is going on here.

```
import matplotlib.pyplot as plt

y=[1,5,25,125]
plt.plot(y)
```

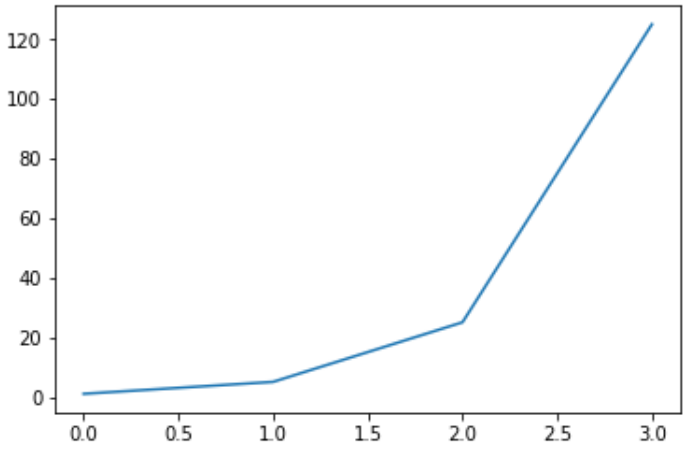

Figure 5: Simple plot 2

Now it is more clear what is happening. Pyplot takes the provided list, plots the elements on the y-axis against the indices of those elements, doing linear interpolation in between. In other words, We are plotting y[i] against i, and interpolating between y[i] and y[i+1].

It is also possible to specify a list as x axis, so we have a list y=[...] and x=[...], and we wish to plot y[i] against x[i]. Maybe we have some time data, and x stores these times while y stores the data points.

Let us try and plot a sine curve. Define an initial time t0, final time tf, and a timestep dt, aka a resolution at which we want to generate data points. Now we use that to generate the time list:

```
import matplotlib.pyplot as plt
import math

t0=0
tf=10
dt=0.1
time=[]
currentTime=t0

while currentTime<=tf:
    time.append(currentTime)
    currentTime+=dt
```

So we keep adding increments of dt to currentTime, until we have passed the value of tf.

Next, we want to get a $y(t) = sin(t)$, so we go through t, and append $sin(t)$ to y:

```
import matplotlib.pyplot as plt
import math

t0=0
tf=10
dt=0.1
time=[]
currentTime=t0

while currentTime<=tf:
    time.append(currentTime)
    currentTime+=dt

y=[]

for t in time:
    output=math.sin(t)
    y.append(output)

plt.plot(time, y)
```

Lastly, we added a plt.plot(time, y), which indicates that time is the x axis, and y is the y axis. This generates the following plot:

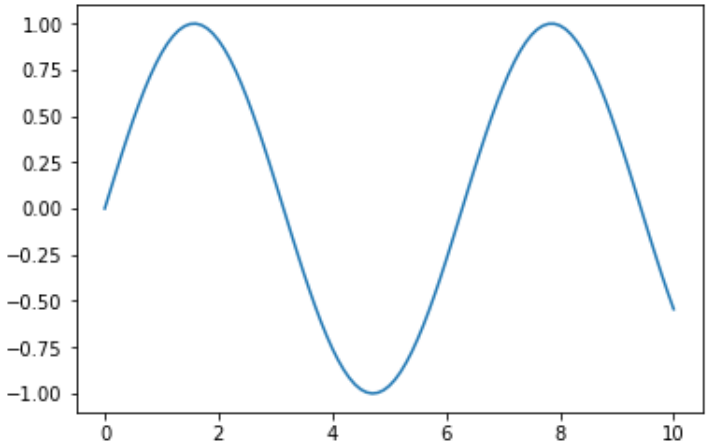

Figure 6: Sin plot

One thing everyone loves about numpy is that almost all other popular modules support numpy arrays, and matplotlib is one of them. Let us perform the above task with the help of numpy:

```
import matplotlib.pyplot as plt
import numpy as np
t0=0
tf=10
dt=0.1
time = np.arange(t0, tf, dt)
y = np.sin(time)
plt.plot(time, y)
```

As you can see, it produces an identical plot, with much fewer lines of code:

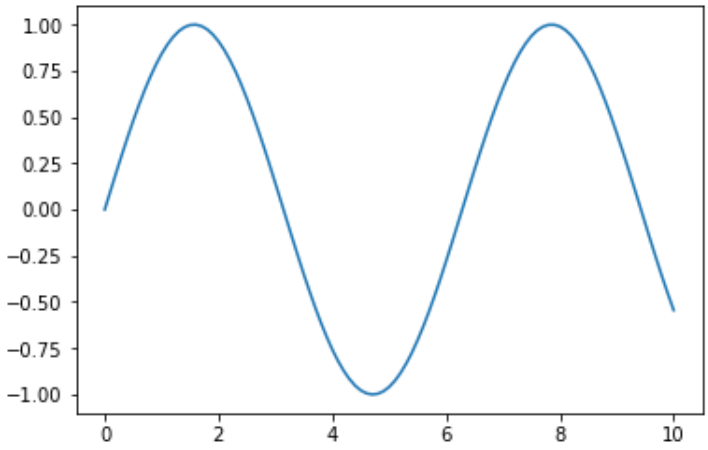

Figure 7: Sin plot using numpy

So np.arrange() is a function that generates an evenly spaces array from t0 to tf with a difference of dt, so np.arrange(t0,tf,dt)=[t0, t0+dt, t0+2dt,...,tf]. Most importantly, matplotlib easily deals with numpy arrays, which is a huge benefit.

For now, our plots have been rather bland and unorganised. For instance we can label our axes and put a legend. Let us now plot out prices. You should by now know how to import and process the csv and store the values in the list "prices". To plot, we use the following code:

```
plt.plot(t, prices, label='Price')
plt.xlabel('time')
plt.ylabel('GBP/USD')
plt.title("GBP to USD")
plt.show()
```

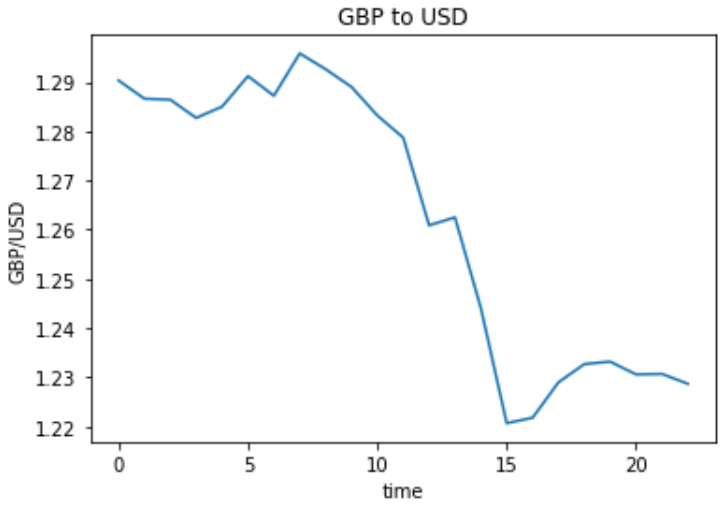

Figure 8: GBP/USD plot

So we used the xlabel and ylabel function to label axes, as well as added a title. Plt.show() is a function to actually display the plots, which can be omitted in jupyter but should generally always be used after doing plots. There are many more functionalities in matplotlib, and on demand can be explored in their documentation.

7.6 Pandas

We have used arrays and lists to store our datasets, and visualised them using matplotlib. However, there is a more convenient way of dealing with csv files and large datasets; that is through use of a module called pandas. Install pandas by running the following command:

```
$ pip3 install pandas
```

Pandas allows the user to create something called Dataframes, which are essentially tables not unlike the ones you have in Excel. Let us import pandas and open the prices csv and read it into a variable:

```
import pandas as pd
import csv

csvfile = open('GBP_USD_data.csv')
readCSV = csv.reader(csvfile, delimiter=',')
```

Next, we can use pandas dataframe function to create a dataframe variable with our data, which we conveniently call "df":

```
df=pd.DataFrame(readCSV)
```

Now we can visualise df by either typing print(df), or for jupyter, simply type df at the end of the code, which will display the dataframe in a nicer looking format. However, our datasets are ofter huge, so a better idea is instead to open just the first few rows of the dataframe, which can be done by calling the head function:

```
df.head()
```

which outputs the following table: We can also directly get a dataframe from a csv:

	0	1
0	Date	Open
1	Oct 31, 2019	1.2903
2	Oct 30, 2019	1.2866
3	Oct 29, 2019	1.2864
4	Oct 28, 2019	1.2827

Figure 9: df.head()

```
import pandas as pd
df= pd.read_csv('GBP_USD_data.csv')
```

Giving the same result.

What we immediately see is that the actual data has included "Date" and "Open" tags, and used custom indexing for the rows and columns. Usually, we wish to index the rows with the first column in our spreadsheet, which is the case for these prices. Then we can simply add an optional parameter to our dataframe initialisation:

```
import pandas as pd

df= pd.read_csv('GBP_USD_data.csv', index_col=0)
df.head()
```

Giving the following output:

Date	Open
Oct 31, 2019	1.2903
Oct 30, 2019	1.2866
Oct 29, 2019	1.2864
Oct 28, 2019	1.2827
Oct 25, 2019	1.2850

Figure 10: dataframe

Let is look at some more complicated data. We will now look at Apple prices for the August-October 2019, but include other parameters such as high, low, volume etc. Load it in and display the head:

```
import pandas as pd
df= pd.read_csv('AAPL.csv', index_col=0)
df.head()
```

Date	Open	High	Low	Close	Adj Close	Volume
2019-08-12	199.619995	202.050003	199.149994	200.479996	200.479996	22481900
2019-08-13	201.020004	212.139999	200.479996	208.970001	208.970001	47218500
2019-08-14	203.160004	206.440002	202.589996	202.750000	202.750000	36547400
2019-08-15	203.460007	205.139999	199.669998	201.740005	201.740005	27227400
2019-08-16	204.279999	207.160004	203.839996	206.500000	206.500000	27620400

Figure 11: AAPL dataframe

Now that the dataframe is set up, we can try and access certain elements. For instance, if we wish to know the data on the 4th of September, we can use the loc function (presumably standing for "locate"):

```
df.loc["2019-09-04"]
```

If we from there further wish to isolate a given column, say volume traded, we simply pass a second argument to the index indicating the column:

```
df.loc["2019-09-04", "Volume"]
```

to give us 19188100.

We can also provide multiple indices as the date, so if we wish to look at only 4th and 5th of September, we can provide a list of arguments:

```
df.loc[["2019-09-04", "2019-09-05"]]
```

Note that the first argument refers to the row, and the second to the column, so we had to index the row providing a list of values, and not separate by a comma. Now we can also do the same for the columns:

```
df.loc[["2019-09-04", "2019-09-05"], ["Open","Volume"]]
```

To give the following dataframe:

Date	Open	Volume
2019-09-04	208.389999	19188100
2019-09-05	212.000000	23913700

Figure 12: AAPL open and volume for 4th and 5th of September

Now we can also plot using df.plot() (function comes from matplotib):

```
df.plot(kind='scatter',x='Open',y='Close',color='blue')
```

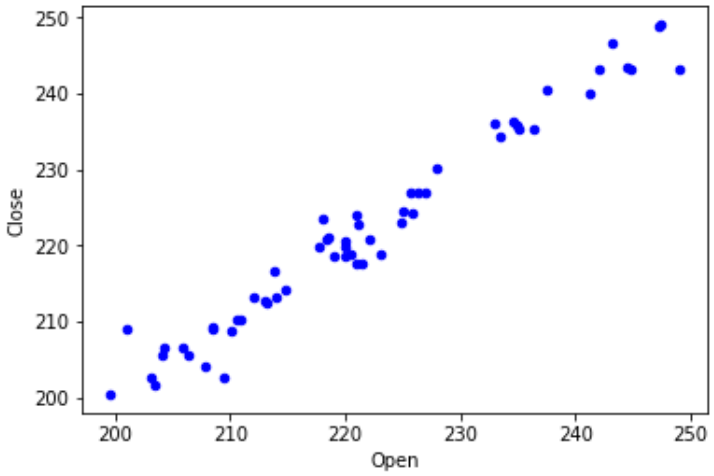

Figure 13: Opening vs closing prices

We can furthermore pull out pieces of data and put them into lists, so for instance we can take out all the opening prices and compute the averages:

```
import statistics as stats
openPrices=df.loc[:, "Open"].tolist()
print( stats.mean(openPrices) )
print( stats.stdev(openPrices) )
```

We said we want all the rows using the colon operator (recall slicing of lists), and indicated "Open" to be the row. On that, we applied the tolist() function, which essentially converts the dataframe object to a list.

We can also go the other way. Suppose you recieve raw input on names, ages and eye colors of different people. We can then save them into a csv using the following code:

```
#raw input
names="Jay, Hannah, Alex"
ages="5, 17, 95"
eyecolor="Blue, Brown, Green"

#process into lists
names=names.strip(" ").split(",")
ages=ages.strip(" ").split(",")
eyecolor=eyecolor.strip(" ").split(",")
people={"Name":names, "Age":ages, "Eye color":eyecolor}

#create dataframe and save to csv
df=pd.DataFrame(people)
df=df.set_index('Name')
df.to_csv("People.csv")
```

We get inputs as strings, so we first use strip() with argument " ", which removes all the whitespaces. Next, we use the split() function with argument "," to split the values into a list. Next, we add all these lists into a dictionary as values, assigning the keys to be "Name", "Age" and "Eye color". Keys will become headers, and values will serve as columns. Then we also set index to be "Names" column, and finally export it to csv using to_csv() function. The dataframe for this dictionary looks as following:

Name	Age	Eye color
Jay	5	Blue
Hannah	17	Brown
Alex	95	Green

Figure 14: Dataframe from dictionary