WORLDQUANT UNIVERSITY

Compiled Content Module 1

MScFE 630 Computational Finance

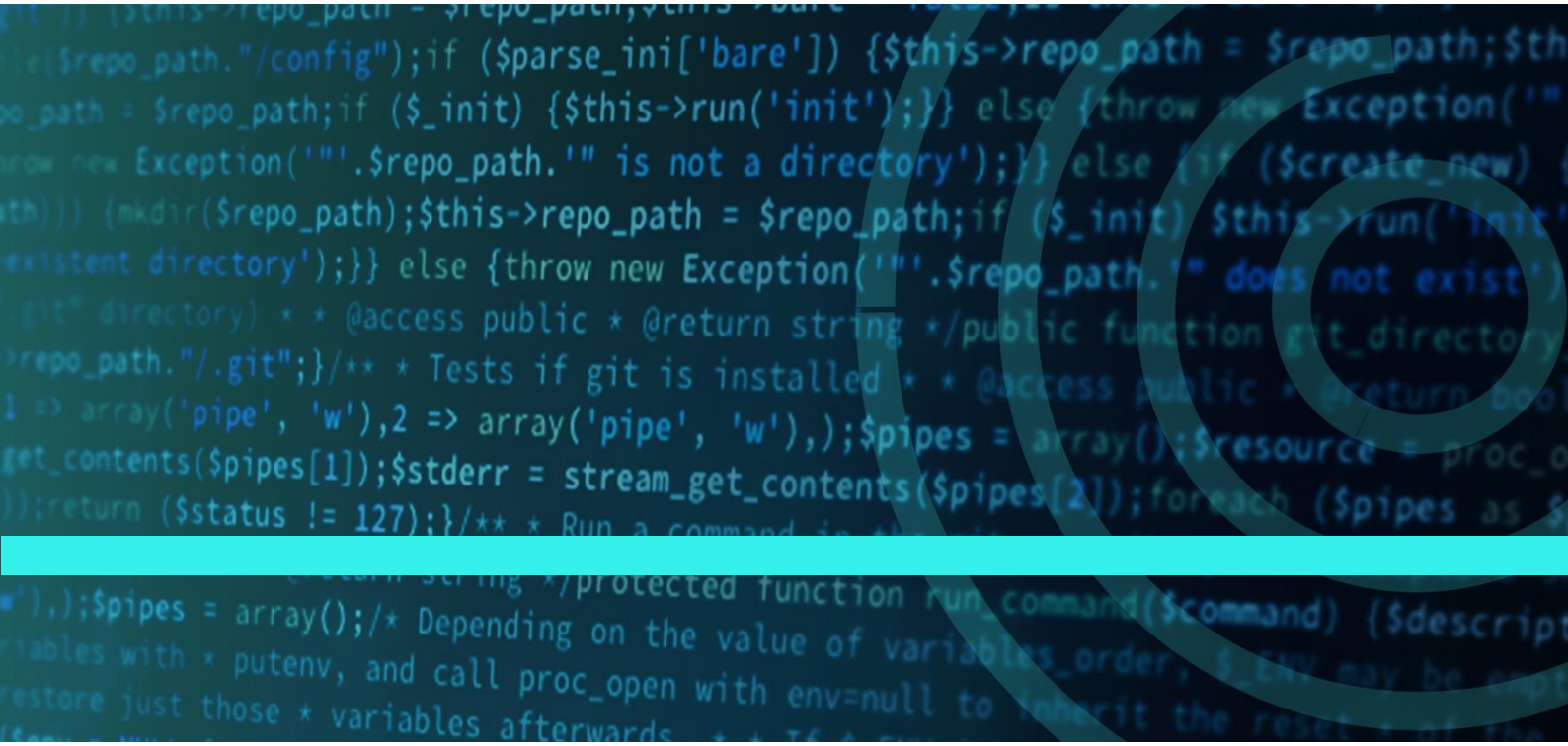
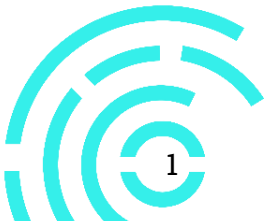


Table of Contents

Revised: 08/19/2019

Module 1: Introduction to Programming in Python

Module 1 of this course introduces Python programming for beginners. The module starts by introducing the building blocks of Python programming through the examination of data types, variables, data structures, loops, and conditional statements. The module continues by explaining how to import external modules and provides an overview of the most common external modules used in computational finance: NumPy, Pandas, SciPy and Matplotlib. The module concludes with an introduction to vectorization..



Unit 1: Getting Started with Python

Unit 1: Video Transcript

Welcome to the first module in our course, Introduction to Programming in Python. In this first video, we will learn to set up a Python work environment, write our first line of Python code, and then learn about the basic data types of Python.

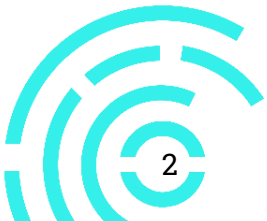
We will be using the Jupyter Notebook as a work environment for the duration of this course. This is an open-source platform for interactive data science and scientific computing. Detailed instructions for installation are available on the Jupyter Notebook website. We encourage you to follow the Anaconda installation method, as this will install crucial modules which we will make use of later in the course. Make sure you download the Python 3.6 version of the Anaconda distribution.

In Jupyter Notebook, we write our code in input cells, which are added using the "+" button in the toolbar. Input cells allow us to separate lines of code into groups which will be executed together. When we select a cell and click the "Run" button, the lines of code in that cell are executed sequentially, and any output from that code is printed below the cell.

```
In [ ]:
```

For now, let's look at a simple example of a Python program. We will write a simple program to output the phrase "Hello World" to the screen.

We can complete this task easily using the print function. A function is simply a predefined block of code which may accept a number of arguments, which is simply data we send to the function. We execute a function by typing its name, followed by a set of arguments in rounded brackets. Even in the case of an empty set of arguments, we still type the rounded brackets.



```
In [1]: print("Hello World")

        Hello World
```

Now that we've created our first simple Python program, let's talk about data types.

In Python, the data we work with is stored as several **types**. These types are used to distinguish between different forms of data. For example, a line of text would be stored as a different type to a numeric value. Types allow us to define different operations which can be performed on different data. Python defines types for both individual data, which we refer to as **data types**, as well as for collections of data, which we refer to as **data structures**.

We will now look at some of the basic data types in Python. These can be divided into three categories:

1. Numeric
2. Strings
3. Boolean

Data types allow us to define operations separately for data which might need to be processed differently. This means that only operations which make sense for a given type of data will be defined for that data type. Let's look at some examples in Jupyter Notebook.

Python's **type** function is useful for determining the data type of a piece of data, if you are unsure.

However, this is primarily a learning tool and is not likely to have practical usage when doing actual work with Python. The output line will reveal what the data type is when using this function.

Let's start looking at some specific data types. We will start by looking at numeric data types. Specifically, these could be integer values, floating-point values, or even complex values.

Integers are whole numbers, without any fractions, and are written as a series of digits.

```
In [1]: type(1)
Out[1]: int
```

Floating-point numbers are numbers which include fractions, and are written as a series of digits representing the whole part, followed by a decimal point and another series of digits representing the fractional part.

```
In [2]: type(3.14)
Out[2]: float
```

Complex numbers are numbers which include an imaginary component and are written as a series of digits representing the real component, plus a series of digits followed by the letter "j", to represent the imaginary component.

```
In [3]: type(1+3j)
Out[3]: complex
```

Now we will discuss Boolean data. The Boolean data type is used to represent a true or false value.

This is used extensively in **conditionals**, which we will discuss in detail in the second section of this module.

```
In [4]: type(True)
Out[4]: bool
```

In Python, we must capitalize "True" and "False" for these to be recognized as Boolean values. If we type these as lowercase instead, an error will occur.

```
---------------------------------------------------------------------
----
NameError                                 Traceback (most recent call l
ast)
<ipython-input-5-6f4d8242c3d0> in <module>()
----> 1 type(true)

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

Finally, let's briefly discuss **string data**. Strings represent series of characters which make up text data, and we write them as text between sets of quotation marks or inverted commas.

```
In [1]: type("This is a string")
Out[1]: str

In [2]: type('This is also a string')
Out[2]: str
```

In this video, we went through the process of installing Python and Jupyter Notebook on your own computer, before taking an introductory look at some of Python's basic syntax and data types.

In the next video, we will cover the concept of variables and their usage in Python programs. Then, we will look at how we can organize data into structures which make it easier to work with large sets of data. Finally, we will explore looping in Python as a way of introducing repetition into our programs without duplicating code.

Unit 1 : Notes

An Introduction to Basic Syntax and Data Types

What is Python?

Python is a programming language which has become popular for general-purpose and scientific programming alike, largely due to its simple syntax and ease of use. Don't let the simplicity of the language mislead you, however. Python can be a powerful tool for quantitative finance when configured with the correct extension modules. This set of notes will cover the process of setting up your own Python work environment, before launching into a crash course on the basics of Python programming. By the end of this section, you should be comfortable writing simple programs using Python.

Setting up your Python work environment

Before you can get started using Python to create useful programs, you will need to set up a work environment to facilitate the writing and running of programs. While there are many ways one could go about this task, the most common for scientific and financial programming is to use the **Jupyter Notebook**. This is an open-source platform for interactive data science and scientific computing. For the duration of the course, it will be expected that you make use of Jupyter Notebook when creating Python code. Instructions for installation on your computer can be found here. We recommend that you follow the Anaconda installation method, as this will also install crucial modules for Computational Finance that will be used later in the course. A guide to getting started with the notebook interface is available here, and reading through this is highly recommended.

Note: It is important to ensure that you download and install the correct version of the Anaconda distribution – we will make use of the **Python 3.6 version**.

Writing your first line of code

In Jupyter Notebook, we write our code in **input cells**, which are added using the "+" button in the toolbar. Input cells allow us to separate lines of code into groups which will be executed together. When we select a cell and click the "Run" button, the lines of code in that cell are executed sequentially and any output from that code is printed below the cell.

Figure 1: Example of an input cell

The classic introductory task in programming is to print out the statement "Hello World". This is a simple problem in Python, requiring only one line of code using the **print** function. A function is simply a predefined block of code which may accept a number of arguments (sometimes referred to as parameters). We execute a function by typing its name followed by a set of arguments in rounded brackets, as shown below.

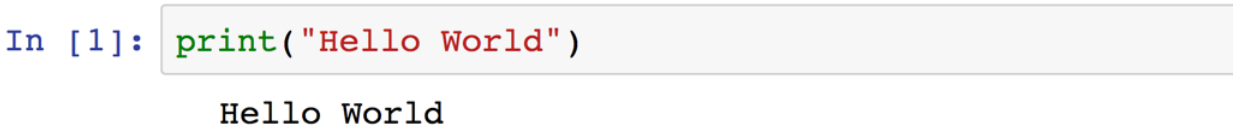


Figure 2: The input cell containing the print statement, as well as the output when this cell is run

Even in the case of an empty set of arguments, we still type the brackets.

Basic data types and operations

Now that you're familiar with adding and running input cells, let's talk about how we can store and use data to make more interesting programs.

In Python, the data we work with is stored as several **types**. These types are used to distinguish between different forms of data. For example, a line of text would be stored as a different type to a numeric value.

Types allow us to define different operations which can be performed on different data. Python defines types for both individual data, which we refer to as **data types**, as well as for collections of data, which we refer to as **data structures**.

We will now look at some of the basic data types in Python. Broadly speaking, basic data types can be divided into three categories:

1. Numeric
2. String
3. Boolean

Data types allow us to perform operations separately for data which might need to be processed differently. This means that only operations which make sense for a given type of data will be defined for that data type. Python's **type** function is useful for determining the datatype of a piece of data, if you are unsure. We will demonstrate its usage below as we go through each of the three categories of data types in detail.

Numeric

Numeric types are used to represent "number data". Specifically, these could be integer values, floating-point values, or even complex values.

Integers are whole numbers, without any fractions, and are written as a series of digits:

```
In [1]: type(1)
Out[1]: int
```

Figure 3: Example of an integer

Floating-point numbers are numbers which include fractions, and are written as a series of digits representing the whole part, followed by a decimal point and another series of digits representing the fractional part:

```
In [2]: type(3.14)
Out[2]: float
```

Figure 4: Example of a floating-point number

Complex Numbers are numbers which include an imaginary component, and are written as a series of digits representing the real part, followed by a "+" symbol, another series of digits representing the imaginary part, and finally the letter "j" to indicate that this is an imaginary component:

```
In [3]: type(1+3j)
Out[3]: complex
```

Figure 5: Example of a complex number

Numeric operations

The basic arithmetic operations are all relatively simple to perform in Python:

1. We **add** numbers using the "+" operator.
2. We **subtract** numbers using the "–" operator.
3. We **multiply** numbers using the "*" operator.
4. We **divide** numbers using the "/" operator.

```
In [1]: 4 + 3
Out[1]: 7

In [2]: 4 - 3
Out[2]: 1

In [3]: 4 * 3
Out[3]: 12

In [4]: 4 / 3
Out[4]: 1.3333333333333333
```

Figure 6: Basic arithmetic operations in Python

However, a few operations might seem a bit more unfamiliar:

1. We **exponentiate** numbers using the "**" operator.
2. We get the remainder of the division of two numbers using the "%" operator, also known as **modulus**.
3. We do **floor division**, which rounds down the result of a normal division to the next integer value, using the "//" operator.

```
In [1]: 2 ** 3
Out[1]: 8

In [2]: 2 % 3
Out[2]: 2

In [3]: 4 // 3
Out[3]: 1
```

Figure 7: More numeric operations in Python

Boolean[1]

The Boolean data type is used to represent a true or false value. This is used extensively in **conditionals**, which we will discuss in detail in the next section.

```
In [4]: type(True)

Out[4]: bool
```

Figure 8: Example of a Boolean value

Note: In Python, we must capitalize "True" and "False" for these to be recognized as Boolean values. If we type "true" or "false" instead, an error will occur:

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-5-6f4d8242c3d0> in <module>()
----> 1 type(true)

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

Figure 9: Error displayed when one doesn't capitalize "True"

Boolean operations

The basic Boolean operations are **and**, **or**, and **not**:

and: This operation checks if both the left and right sides are true – and, if so, it outputs true; otherwise, it outputs false.

[1] Technically, at a low level, Boolean values are stored as integers. However, we differentiate them here because semantically they are very different.

```
In [1]: True and True
Out[1]: True

In [2]: True and False
Out[2]: False

In [3]: False and False
Out[3]: False
```

Figure 10: Example of the usage of the "and" operation in Python.

or: This operation checks if either left or right (or both) are true – and, if so, it outputs true; otherwise it outputs false.

```
In [4]: True or True
Out[4]: True

In [5]: True or False
Out[5]: True

In [6]: False or False
Out[6]: False
```

Figure 11: Example of the usage of the "or" operation in Python.

not: This operation inverts the following Boolean value. If the following value is true, it outputs false, and vice versa.

```
In [7]: not True
Out[7]: False

In [8]: not False
Out[8]: True
```

Figure 12: Example of the usage of the "not" operation in Python.

Strings

Strings represent series of characters which make up text data, and we write them as text between sets of quotation marks or inverted commas.

```
In [1]: type("This is a string")
Out[1]: str

In [2]: type('This is also a string')
Out[2]: str
```

Figure 13: Examples of string values in Python

String operations

There are multiple useful operations that Python affords us in order to work with string data:

Concatenation is done using the "+" operator. This allows us to join two strings together.

```
In [1]: "Hello" + " " + "World" + "!"
Out[1]: 'Hello World!'
```

Figure 14: Example of the usage of the concatenation operation in Python

Repetition is done using the "*" operator. This allows us to repeat a string many times without explicitly typing it out.

```
In [2]: "Repetition! " * 5
Out[2]: 'Repetition! Repetition! Repetition! Repetition! Repetition! '
```

Figure 15: Example of the usage of the repetition operation in Python

String indexing is done using square brackets with some integer value inside. This gives us a string containing the single character at the index, denoted by the integer value.

Note: All indexing in Python starts from 0, not from 1. This is common practice in many programming languages.

```
In [3]: "Hello" [1]
Out[3]: 'e'
```

Figure 16: Example of the usage of the character slice operation in Python

String slicing is also done using square brackets, but with two integer values separated by a colon inside. This gives us the string of characters from the first index to the second (inclusive of the first index, but exclusive of the second).

```
In [4]: "Hello"[1:5]
Out[4]: 'ello'
```

Figure 17: Example of the usage of the range slicing operation in Python

Membership is done using the **in** operator. This gives us a Boolean value representing whether or not the first string is contained in the second.

```
In [5]: "Hello" in "Hello World!"
Out[5]: True

In [6]: "Goodbye" in "Hello World!"
Out[6]: False
```

Figure 18: Example of the usage of the membership operation in Python

Getting help

Many of the concepts which have been introduced in these notes (and indeed, many of those which will be covered later in this module) have very detailed guides to their usage in the official Python documentation. This documentation can be accessed at https://docs.python.org/3/index.html

Conclusion

In this section, we went through the process of installing Python and Jupyter Notebook on your own computer, before taking an introductory look at some of Python's basic syntax. We then covered the basic data types of Python and looked at some of the specific operations which Python allows us to perform on this data.

In the next section, we will cover the concept of variables and their usage in Python programs. Then, we will look at how we can organize data into structures, which will make it easier to work with large sets of data. Finally, we will explore looping in Python as a way of introducing repetition into our programs without duplicating code.

.

Unit 2: Further Concepts in Python

Unit 2 : Video Transcript

Welcome to the second video of the Introduction to Programming in Python module. In this video, we will begin by discussing the concept of variables and their usage in Python. We will then move on to an explanation of basic data structures and loops in Python.

Up until this point, the only data we have interacted with has been in the form of literals, which are simply data values which cannot be changed. We can now, however, begin to look at **variables**. These are essentially labels that we assign to data, which allow us to reuse and modify that data throughout a program. This is very similar to the concept of variables which is introduced in basic algebra classes.

In Python, there are a few simple rules for the naming of variables:

- Variable names must start with either a letter or an underscore, "_".
- Subsequent characters may be made up of a combination of letters, numbers or underscores.
- Variable names are case sensitive ("I" is not the same as "i").
- Variable names may not contain spaces, as these are used to determine the end of a variable's name.

For example, these variable names would be considered valid by Python's standards:

☑ number_of_dogs

☑ ground0

☑ stock_Price

But these variable names would not be valid:

- ☒ number of dogs (contains spaces)
- ☒ 0ground (starts with a number)
- ☒ stock–Price (contains an invalid character, ' – ')

Now, let's look at how we can use variables in code.

To assign a value to a variable, we type the variable's name, followed by the "=" symbol, followed by the value we wish to assign to the variable. Take special note of this order, as reversing it will result in errors. The left-hand side must always be the variable to which the value is being assigned.

```
In [1]: name = "Leo Messi"
        age = 30
        height = 1.7
        occupation = "Football Player"
        is_a_forward = True
```

If we don't adhere to this order, we will encounter errors when trying to run our code.

```
In [2]: 20 = age
          File "<ipython-input-2-4672a15b62f3>", line 1
            20 = age
               ^
        SyntaxError: can't assign to literal
```

Once we have assigned a value to a variable, we can use it in the same way that we have used literal values previously.

```
In [1]: x = 1
        x

Out[1]: 1

In [2]: a = 2
        b = 3
        a+b

Out[2]: 5

In [3]: name = "Harry"
        surname = "Potter"
        name + " " + surname

Out[3]: 'Harry Potter'

In [4]: has_a_scar = True
        wears_glasses = True
        has_blonde_hair = False

In [5]: has_a_scar and wears_glasses

Out[5]: True

In [6]: has_a_scar and has_blonde_hair

Out[6]: False

In [7]: has_a_scar or has_blonde_hair

Out[7]: True

In [8]: not wears_glasses

Out[8]: False
```

The primary difference between using variables and literals is that we can modify a variable's value by reassigning it; and the name of a variable affords it additional semantic meaning, which helps to make the code more readable.

```
In [3]: x = 4

        x = 5
        x

Out[3]: 5
```


While we have now covered most of the tools required to write data-driven programs in Python, it would be very difficult to write programs which deal with large datasets without some way organizing them. That's where **data structures** come in. Data structures are ways of storing, retrieving and operating on, data, thus making it easier to work with substantial amounts of data. We will cover two very common Python data structures in this video:

1. Lists
2. Tuples

Lists are the most commonly used data structure in Python. They are very flexible, allowing us to append and remove elements as needed using the **append** and **remove** functions, as well as the **del** keyword.

```
In [4]: x.remove(4)
        x

Out[4]: [1, 3, 5]
```

```
In [1]: x = [1, 2, 3]
        x

Out[1]: [1, 2, 3]

In [2]: del x[1]
        x

Out[2]: [1, 3]

In [3]: x.append(4)
        x.append(5)
        x

Out[3]: [1, 3, 4, 5]
```

They are also type-agnostic, which means that one list can contain data of multiple different types.

```
In [5]: values = ["lionel", "messi", 30, True]
        values

Out[5]: ['lionel', 'messi', 30, True]
```


Tuples are similar to lists but cannot be modified as easily. These will be covered in detail in the notes that follow.

Looping

There are times when we need to repeat a set of steps many times over, but it wouldn't be very practical to type these repeated lines out. Rather, Python affords us the ability to construct loops. These are blocks of code which can repeat many times depending on some condition.

There are two types of loops in Python:

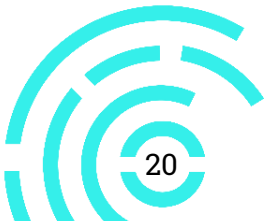
1. **For** loops
2. **While** loops

For loops are used for when you want to repeat a block of code a set number of times (this number can be predetermined or based on a variable such as the size of a dataset).

All loops require us to indent the code that we want to fall within the scope of the loop. We can use the "tab" key to do this. Once we have reached the end of the code which we want to fall within the loop structure, we must stop indenting subsequent lines.

We tend to make extensive use of the **range** function in **for loops**. This function takes at least one, but up to three parameters:

1. If only one parameter is given, it defines the number of integers to generate starting from zero. The start value is assumed to be 0 and the step value (the number added to the current value after each iteration of the loop) is assumed to be either 1 or -1.



In [1]:
```
for x in range(10): # x will start at the value 0, increasing by 1
    # each iteration until it reaches the value 10
    print(x) # note the indentation of this line resulting in this code
    # being repeated

print("This is not in the loop") # note that, because this line isn't
# indented, it won't be repeated
```

```
0
1
2
3
4
5
6
7
8
9
This is not in the loop
```

2 If two parameters are given, the first parameter defines the start value and the second defines the excluded end value. The step value is again assumed to be either 1 or -1.

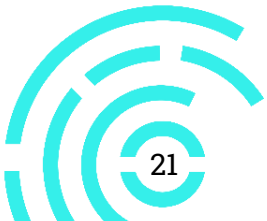
In [2]:
```
for x in range(5, 10): # now x will start at the value 5, increasing by 1
    # each iteration until it reaches the value 10
    print(x)
```

```
5
6
7
8
9
```

3 If all three parameters are given, the first two parameters function as above, but the third defines the step value.

In [3]:
```
for x in range(5, 10, 2): # now x will start at the value 5, increasing
    # by 2 each iteration until it reaches the value 10
    print(x)
```

```
5
7
9
```

We can also use a for loop to directly access the data in a data structure, by simply replacing the range function call with the variable name to which that data structure is assigned.

```
In [4]: teams = ["Chelsea", "Manchester United", "Arsenal", "Leeds United"]

        for team in teams:
            print(team)
```

```
Chelsea
Manchester United
Arsenal
Leeds United
```

While loops are loops which keep iterating until a Boolean condition evaluates to false. These will be covered in detail in the notes that follow.

In this video, we learned about variables and how to use them in Python programs, before briefly going through Python's list data structure. We then covered looping and looked at how we can use for loops to introduce repetition into our programs.

In the next video, we will look at conditional statements and how they allow us to introduce branching behaviour into our programs, before taking a deeper look at the concept of functions. We will then look at importing external modules into Python, and explore a particularly useful module for computational finance.

Unit 2: Notes

An Exploration of Variables, Commenting, Data Structures and Loops

Understanding variables

Up until this point, the only data we have interacted with has been in the form of literals, which are simply data values which cannot be changed. Now, we will begin to consider variables. These are essentially the labels we assign to data, which allow us to reuse and modify that data throughout a program. This is very similar to the concept of variables which is introduced in basic algebra classes.

Naming variables

In Python, there are a few simple rules for naming of variables:

- Variable names must start with either a letter or an underscore, "_".
- Subsequent characters may be made up of only letters, numbers or underscores.
- Variable names are case sensitive ("I" is not the same as "i").
- Variable names may not contain spaces, as these are used to determine the end of a variable's name.

For example, these variable names would be considered valid by Python's standards:

☑ number_of_dogs

☑ ground0

☑ stock_Price

But these variable names would not be valid:

- ☒ number of dogs (contains spaces)
- ☒ 0ground (starts with a number)
- ☒ stock–Price (contains an invalid character, ' – ')

Assigning values to variables

To assign a value to a variable, we type the variable's name, followed by the "=" symbol, followed by the value we wish to assign to the variable. Take special note of this order, as reversing it will result in errors. The left-hand side must always be the variable to which the value is being assigned.

```
In [1]: name = "Leo Messi"
        age = 30
        height = 1.7
        occupation = "Football Player"
        is_a_forward = True
```

Figure 19: Example of assigning values to some variables

If we don't adhere to this order, we will encounter errors when trying to run our code.

```
In [2]: 20 = age
          File "<ipython-input-2-4672a15b62f3>", line 1
            20 = age
               ^
        SyntaxError: can't assign to literal
```

Using variables in code

Once we have assigned a value to a variable, we can use it in the same way as we have used literal values previously.

```
In [1]: x = 1
        x

Out[1]: 1

In [2]: a = 2
        b = 3
        a+b

Out[2]: 5

In [3]: name = "Harry"
        surname = "Potter"
        name + " " + surname

Out[3]: 'Harry Potter'

In [4]: has_a_scar = True
        wears_glasses = True
        has_blonde_hair = False

In [5]: has_a_scar and wears_glasses

Out[5]: True

In [6]: has_a_scar and has_blonde_hair

Out[6]: False

In [7]: has_a_scar or has_blonde_hair

Out[7]: True

In [8]: not wears_glasses

Out[8]: False
```

Figure 20: Examples of using variables instead of literals

The primary difference between using variables and literals is that we can modify a variable's value by reassigning it, and the name of a variable affords it additional semantic meaning which helps to make code more readable.

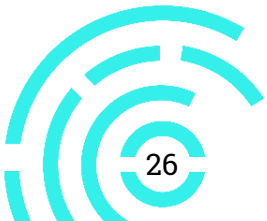
```
In [3]: x = 4

        x = 5
        x

Out[3]: 5
```

Figure 21: Example of reassigning a variable value

While it is possible to simply use a variable's own value to update itself (say, if you had an integer value which you wanted to increment), there is also the option to use shorthand operators built into Python which allow us to do this more succinctly. Operator-equals combinations give us a shorthand for updating a variable by combining its value with another. For example, in the case of the "+=" operator, we use the statement "x += y" in place of the clumsier "x = x + y". Similar combinations are available for many other operators.

```
In [1]: a = 4
        a += 6
        a

Out[1]: 10

In [2]: b = 2
        b -= 3
        b

Out[2]: -1

In [3]: c = 2
        c *= 3
        c

Out[3]: 6

In [4]: d = 4
        d /= 2
        d

Out[4]: 2.0
```

Figure 22: Examples of the use of shorthand operator-equals combinations



A note on commenting

In Python, we can leave comments in our code using the hash symbol, "#". Essentially, this symbol signals that the rest of that line of code is only used for us to comment on our code, and should be ignored entirely by the computer when executing the program.

```
In [1]: # this is a comment, and will not run as code
```

Figure 23: Example of a simple comment

Even if we type in valid Python code after placing a hash, it will not be run. Commenting is a very useful tool for clarifying the intention of our code. This is useful not only to preserve our own understanding of the code we have written, but also so that our code can be understood by other people when collaborating.

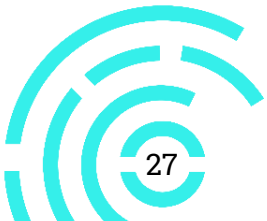
```
In [1]: a = "Commented code is not executed!"
        # a = "Commented code IS executed!"
        a

Out[1]: 'Commented code is not executed!'
```

Figure 24: Example demonstrating that even valid code is not executed when commented out

Basic data structures

While we have now covered most of the tools required to write data driven programs in Python, it would be very difficult to write programs which deal with large datasets without having some way to organize them. That's where **data structures** come in. Data structures are ways of storing, retrieving, and operating on, data, which make it easier to work with substantial amounts of data. We will cover two very common Python data structures in this section:

1. Lists
2. Tuples



Each data structure may have its own benefits, drawbacks and syntax. As such, it is important to note that there is no single "best" data structure. Rather, we must think about the problem at hand and consider which data structure best suits the situation.

Lists

Lists are the most commonly used data structure in Python. They are very flexible, allowing us to append and remove elements as needed using the **append** and **remove** functions, as well as the **del** keyword. When declaring lists, we type a comma-separated series of values and enclose these in square brackets, as seen in the examples below.

```
In [1]: x = [1, 2, 3]
        x
Out[1]: [1, 2, 3]

In [2]: del x[1]
        x
Out[2]: [1, 3]

In [3]: x.append(4)
        x.append(5)
        x
Out[3]: [1, 3, 4, 5]

In [4]: x.remove(4)
        x
Out[4]: [1, 3, 5]
```

Figure 25: Examples of defining a list and performing basic functions

They are also type-agnostic, meaning that one list can contain data of multiple different types.

```
In [5]: values = ["lionel", "messi", 30, True]
        values
Out[5]: ['lionel', 'messi', 30, True]
```

Figure 26: Example demonstrating type agnostic nature of lists

Tuples[2]

Tuples are similar to lists but are immutable. This means that once a tuple has been created, it cannot be changed in any way. Deleting and modifying elements of a tuple is not supported, and if we add an element to a tuple after its creation, a whole new tuple must be generated. This makes tuples less flexible, but generally more efficient, than lists. When declaring tuples, we use very similar syntax to that of declaring lists but, rather than square brackets, we use round brackets.

```
In [1]: ages = (30, 21, 17, 32)
        ages
Out[1]: (30, 21, 17, 32)
```

Figure 27: Example of a tuple declaration

[2] These are actually stored as the same underlying data type as strings, known as the "immutable sequence". That being said, the way we use strings and tuples is very different and so we make this distinction.

```
In [1]: characters = ("Harry Potter", "Hermione Granger", "James Bond", "Ron Weasley")
        del characters[3]
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-1-ed127c473d23> in <module>()
      1 characters = ("Harry Potter", "Hermione Granger", "James Bond", "Ron Weasley")
----> 2 del characters[3]

TypeError: 'tuple' object doesn't support item deletion
```

```
In [2]: characters = ("Harry Potter", "Hermione Granger", "James Bond", "Ron Weasley")
        characters.remove("James Bond")
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-2-8dbf5d0cbe0d> in <module>()
      1 characters = ("Harry Potter", "Hermione Granger", "James Bond", "Ron Weasley")
----> 2 characters.remove("James Bond")

AttributeError: 'tuple' object has no attribute 'remove'
```

```
In [3]: characters = ("Harry Potter", "Hermione Granger", "James Bond", "Ron Weasley")
        characters.append("Neville Longbottom")
        characters
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-3-21ebf0a2876b> in <module>()
      1 characters = ("Harry Potter", "Hermione Granger", "James Bond", "Ron Weasley")
----> 2 characters.append("Neville Longbottom")
      3 characters

AttributeError: 'tuple' object has no attribute 'append'
```

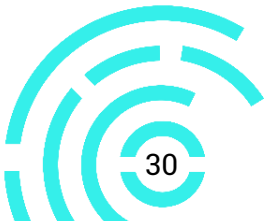
Figure 28: Examples of invalid tuple operations

```
In [4]: characters = ("Harry Potter", "Hermione Granger", "James Bond", "Ron Weasley")
        characters = characters + ("Neville Longbottom",) # this creates an entirely new tuple,
        # which must be reassigned
        characters
```

```
Out[4]: ('Harry Potter',
         'Hermione Granger',
         'James Bond',
         'Ron Weasley',
         'Neville Longbottom')
```

Figure 29: Example of correct tuple concatenation

It is possible to get a list from a tuple using the **list** function. This creates a new list containing all elements from a given tuple.



```
In [5]: characters = ("Harry Potter", "Hermione Granger", "James Bond", "Ron Weasley")
        characters_list = list(characters)
        characters_list

Out[5]: ['Harry Potter', 'Hermione Granger', 'James Bond', 'Ron Weasley']

In [6]: characters_list.append("Neville Longbottom")
        del characters_list[3]
        characters_list

Out[6]: ['Harry Potter', 'Hermione Granger', 'James Bond', 'Neville Longbottom']
```

Figure 30: Using the list method in order to convert a tuple

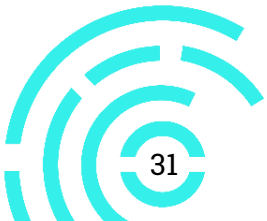
Loops

There are times when we need to repeat a set of steps many times over, but it wouldn't be very practical to keep typing these repeated lines out. Rather, Python affords us the ability to construct loops. These are blocks of code which can repeat many times over, depending on some condition. Loops require us to indent the code that we want to fall within the scope of the loop. We can use the "tab" key to do this. Once we have reached the end of the code which we want to fall within the loop structure, we must stop indenting subsequent lines.

There are two types of loops:

For loops

These loops are used for when you want to repeat a block of code a set number of times (this number can be predetermined or based on a variable such as the size of a dataset). We tend to make extensive use of the range function in for loops. This function takes at least one, but up to three parameters.



If only one parameter is given, it defines the number of integers to generate starting from zero. The start value is assumed to be 0 and the step value is assumed to be either 1 or -1.

```
In [1]: for x in range(10): # x will start at the value 0, increasing by 1
            # each iteration until it reaches the value 10
            print(x) # note the indentation of this line resulting in this code
            # being repeated

        print("This is not in the loop") # note that, because this line isn't
        # indented, it won't be repeated
```

```
0
1
2
3
4
5
6
7
8
9
This is not in the loop
```

Figure 31: Example of using a ranged for loop with one parameter

If two parameters are given, the first parameter defines the start value and the second defines the excluded end value. The step value is again assumed to be either 1 or -1.

```
In [2]: for x in range(5, 10): # now x will start at the value 5, increasing by 1
            # each iteration until it reaches the value 10
            print(x)
```

```
5
6
7
8
9
```

Figure 32: Example of using a ranged for loop with two parameters

If all three parameters are given, the first two parameters function as above, but the third defines the step value.

```
In [3]: for x in range(5, 10, 2): # now x will start at the value 5, increasing
            # by 2 each iteration until it reaches the value 10
            print(x)

5
7
9
```

Figure 33: Example of using a ranged for loop with three parameters

We can also use a for loop to directly access the data in a data structure, by simply replacing the range function call with the variable name to which that data structure is assigned.

```
In [4]: teams = ["Chelsea", "Manchester United", "Arsenal", "Leeds United"]

        for team in teams:
            print(team)

Chelsea
Manchester United
Arsenal
Leeds United
```

Figure 34: Example of looping through the items in a list

While loops

These loops are used when you want to repeat a block of code until a Boolean condition evaluates to false. Often, we make use of comparison operators to generate Boolean values by comparing the values of other variables.

Comparison Operator	Meaning	Explanation
==	'equals'	Checks if the left value is equal to the right value.
!=	'not equals'	Checks if the left value is not equal to the right value.
>	'greater than'	Checks if the left value is strictly greater than the right value.
>=	'greater than or equal to'	Checks if the left value is greater than or equal to the right value.
<	'less than'	Checks if the left value is strictly less than the right value.
<=	'less than or equal to'	Checks if the left value is less than or equal to the right value.

```
In [5]: x = 5

        while (x > 0):
            print(x)
            x -= 1
```

```
5
4
3
2
1
```

Figure 35: Example of a valid while loop

We must be very careful when using these types of loops, because, if the Boolean condition never evaluates to false, then the program will be stuck in an infinite loop.

```
In [6]: x = 5

        while (x > 0):
            print(x)
            x+=1

5
6
7
8
9
10
11
12
13
14
15
16
```

Figure 36: Example of a while loop declaration which results in infinite repetition

Conclusion

In this section, we explored variables and their syntax in Python before briefly covering commenting and its importance in generating easily readable and understandable code. We then went through the concept of data structures and looked specifically at two of the most common data structures which are built into Python: lists and tuples. We then covered looping with for and while loops, which allow us to build repetition into our programs without copying code.

In the next section, we will look at how we can build branching functionality into our programs using conditional statements, before taking a deeper look at functions and how we can declare our own functions in Python. Then, we will cover importing external modules into Python, which allow us to expand upon Python's built-in functionality.

Unit 3: Conditionals, Functions and External Modules

Unit 3: Video Transcript

Hi and welcome to the third video in this module. In this video, we will begin by discussing conditional statements and how we can use these to introduce branching behaviour in our programs. We will then take a deeper look at functions before learning to how to import external modules into Python programs. Finally, we will cover NumPy, which is an important external module used in computational finance.

Let's start by talking about **conditional statements**. These are special statements which allow us to provide different functionality depending on some Boolean value. There are two types of conditional statements: **If** statements and **Else** statements.

If statements check a Boolean value and, if it evaluates to true, execute the code indented after the statement. If the value evaluates to false, the block of code is completely skipped.

```
In [1]: if (True):
            print("The 'True' value means this code executes!")
        if (False):
            print("The 'False' value means this code doesn't execute...")
```

```
The 'True' value means this code executes!
```

Else statements must follow from an **if** statement. If that statement is not executed (i.e. its Boolean value evaluates to false), only then will the **else** statement be triggered, and the indented code following the statement will be executed.

In [1]:
```
if (True):
    print("The 'True' value means this code executes!")
if (False):
    print("The 'False' value means this code doesn't execute...")
```

```
The 'True' value means this code executes!
```

In [2]:
```
boolean_val = True

if (boolean_val):
    print("First statement evaluates to true!")
else:
    print("The first statement evaluated to false...")
```

```
First statement evaluates to true!
```

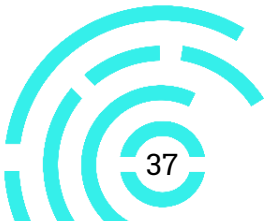
Python also defines a third type of conditional statement, **elif**, but this is simply a contracted combination of else and if. It acts similarly to a normal **if** statement, but its Boolean value is only checked if the previous **if** statement evaluates to false.

In [4]:
```
boolean_val1 = False
boolean_val2 = True

if (boolean_val1):
    print("")
elif (boolean_val2):
    print("The first statement was False and the second was True!")
```

```
The first statement was False and the second was True!
```

In [5]:
```
boolean_val1 = True
boolean_val2 = True

if (boolean_val1):
    print("The first value is True!")
elif (boolean_val2):
    print("This is not printed because the first value was True")
```

```
The first value is True!
```

We will now look at **functions**. We have already introduced the concept of functions in previous sections, but only at a very surface level. Now, we will look at how we can define and implement our own functions.

To define a function in Python, we use the **def** keyword, followed by a function name (using the same naming conventions that we use for variable names), a set of parameters in round brackets and a colon. The function body must be indented in the same way as is done for loops and conditional statements.

In [1]:
```
def hello_world():
    print("Hello World!")

# Nothing is printed out now, because we only declared the function
# and did not call it.
```

In [2]:
```
hello_world()

# Now something will be printed, because we call the function!
```

```
Hello World!
```

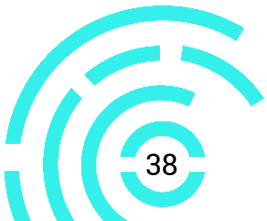
Sometimes, functions are required to calculate a value during execution and send this back to where they were called from. We do this using the **return** keyword.

In [2]:
```
def add_two_numbers(number1, number2):
    return number1 + number2

print(add_two_numbers(1,2))
```

```
3
```

External Modules

Let's discuss external modules. These are packages of code that provide additional functionality over that of the standard Python language.



These modules are very useful as they allow Python users to simply import extra functionality, assuming a suitable module exists, rather than having to write this code themselves. Often external modules will introduce their own data types as well as functions. We will now go through the syntax for importing modules.

We import modules using the **import** keyword followed by the module name. This gives the current program access to all of the special data types and functions introduced in that module.

In [1]:
```
import random # This gives us access to all the functions in the 'random'
# module
```

We can also set an alias for the imported module. This is useful for situations when typing the full name of the module out to refer to its functions and data types would be tedious.

In [2]:
```
import random as rand # Now we can refer to random as 'rand' instead
```

We can then call functions contained in an imported module by typing the name or alias of that module, followed by a dot (".") and the name of the function we are trying to call.

In [3]:
```
for i in range (5):
    x = rand.randint(0, 5) # This returns a random integer between 0 and 5
    print(x)
```

```
4
2
3
3
5
```


Introducing NumPy

We will now look at NumPy, an important external module for Computational Finance. This module introduces a new data structure, the **NumPy array**, along with some operations to make working with these arrays easier. Compared to Python lists, which we covered in the previous section, NumPy arrays have a few differences:

- NumPy arrays contain only a single type of data, compared to Python lists, which can be made up of multiple distinct data types.
- NumPy arrays allow for more efficient operations due to the concept of **vectorization**, which we will cover in detail in the next section. This makes NumPy arrays much more suitable when working with large datasets.

NumPy is simple to import:

```
In [1]: import numpy # This gives us access to the numpy module
```

We can create a NumPy array by calling the module's **array** function, and passing in a list of values with which to initialize the array. This function returns a newly created NumPy array.

```
In [2]: numbers = numpy.array([1,2,3,4,5])
        print(numbers)
```

```
[1 2 3 4 5]
```

We can also create matrices by calling the **array** function with a list of lists.

```
In [3]: matrix = numpy.array([[0, 1, 2], [3, 4, 5], [6, 7, 8]])
        print(matrix)
```

```
[[0 1 2]
 [3 4 5]
 [6 7 8]]
```

We can use the same square bracket notation of Python lists to access specific items in a NumPy array.

We can also use this notation to access particular rows or columns of a matrix.

```
In [8]: numbers = numpy.array([1,2,3,4,5])
        print(numbers[2])

        3
```

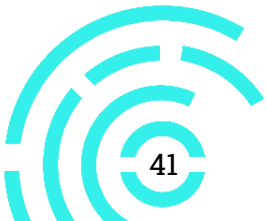
```
In [9]: matrix = numpy.array([[0, 1, 2], [3, 4, 5], [6, 7, 8]])
        print(matrix[0,:]) # access the entire first row

        [0 1 2]
```

```
In [10]: matrix = numpy.array([[0, 1, 2], [3, 4, 5], [6, 7, 8]])
         print(matrix[:,1]) # access the entire second column

         [1 4 7]
```

There are many useful functions for working with these arrays, some of which we will cover in the following set of notes.

In this video, we covered conditional statements, took a deeper look at functions and learned how to declare our own functions. We also learned to import external modules to expand Python's functionality and then looked specifically at the functionality offered by the NumPy module.

In the next video, we will look at three more crucial external modules – namely, Pandas, SciPy and Matplotlib – and learn how to import and use these. We will then go over the concept of vectorization in detail.



Unit 3 : Notes

In these notes, we will introduce branching functionality and offer a guide for importing external modules to expand upon Python's core functionality.

Conditional statements

There are times when we want our programs to provide different functionality depending on some condition. This is when the Boolean data type and comparison operators we discussed previously become very important. You will remember that Boolean data represents true and false values. We can use these, along with **conditional statements**, to allow branching functionality in our programs. There are two types of conditional statements:

If statements

These statements check a Boolean value and, if it evaluates to true, execute the code indented after the statement. If the value evaluates to false, the block of code is completely skipped.

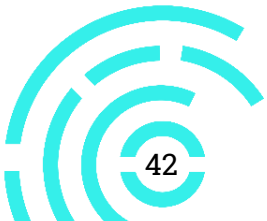
```
In [1]: if (True):
            print("The 'True' value means this code executes!")
        if (False):
            print("The 'False' value means this code doesn't execute...")
```

```
The 'True' value means this code executes!
```

Figure 37: Example demonstrating the use of if statements

Else statements

These statements must follow from an **if** statement. If that statement is not executed (i.e. its Boolean value evaluates to false), only then will the **else** statement be triggered, and the indented code following the statement will be executed.



```
In [1]: if (True):
            print("The 'True' value means this code executes!")
        if (False):
            print("The 'False' value means this code doesn't execute...")
```

```
The 'True' value means this code executes!
```

```
In [2]: boolean_val = True

        if (boolean_val):
            print("First statement evaluates to true!")
        else:
            print("The first statement evaluated to false...")
```

```
First statement evaluates to true!
```

Figure 38: Examples demonstrating the usage of else statements

Technically, Python also defines a third type of conditional statement, **elif**, but this is simply a contracted combination of **else** and **if**. It acts similarly to a normal **if** statement, but its Boolean value is only checked if the previous **if** statement evaluates to false.

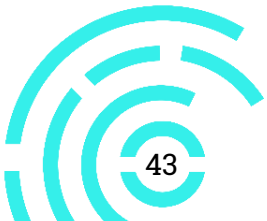
```
In [4]: boolean_val1 = False
        boolean_val2 = True

        if (boolean_val1):
            print("")
        elif (boolean_val2):
            print("The first statement was False and the second was True!")
```

```
The first statement was False and the second was True!
```

```
In [5]: boolean_val1 = True
        boolean_val2 = True

        if (boolean_val1):
            print("The first value is True!")
        elif (boolean_val2):
            print("This is not printed because the first value was True")
```

```
The first value is True!
```

Figure 39: Examples demonstrating the usage of elif statements



A deeper look at functions

We have already introduced the concept of functions in previous sections, but only at a very superficial level. Now, we will look at how you can define and implement your own functions. To define a function in Python, we use the **def** keyword, followed by a function name (following the same naming conventions as for variable names), a set of parameters in round brackets, and a colon. The function body must be indented in the same way as it is done for loops and conditional statements.

```
In [1]: def hello_world():
            print("Hello World!")

        # Nothing is printed out now, because we only declared the function
        # and did not call it.
```

```
In [2]: hello_world()

        # Now something will be printed, because we call the function!
```

```
Hello World!
```

Figure 40: Example of declaring a simple function with no parameters

```
In [1]: def say_hello_to(name):
            print("Hello, " + name + "!")

        say_hello_to("Harry")
```

```
Hello, Harry!
```

Figure 41: Example of declaring a simple function with one parameter

Sometimes, functions are required to calculate a value during execution and send this back to where they were called from. We do this using the **return** keyword.

```
In [2]: def add_two_numbers(number1, number2):
            return number1 + number2

        print(add_two_numbers(1,2))
```

```
3
```

Figure 42: Example of returning a value from a function

Many other functions do not have to return any value, and sometimes these are referred to as "void" functions. One example of such a function is the **print** function which we have used in the past. Its sole purpose is to output text and therefore it does not return any value.

What are external modules?

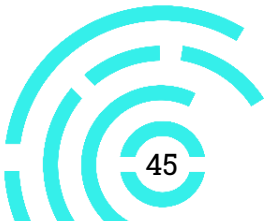
External modules are importable collections of code that provide additional functionality over that of the standard Python language. These modules are very useful as they allow Python users to simply import extra functionality, assuming a suitable module exists, rather than having to write this code themselves. Often external modules will introduce their own data types as well as functions. We will now go through the syntax for importing modules, before exploring a few modules which will be crucial in later parts of this course.

Importing useful modules

We import modules using the **import** keyword followed by the module name. This gives the current program access to all the special data types and functions introduced in that module.

```
In [1]: import random # This gives us access to all the functions in the 'random'
        # module
```

Figure 43: Example of importing a module



We can also set an alias for the imported module. This is useful for situations when typing out the full name of the module to refer to its functions and data types would be tedious.

```
In [2]: import random as rand # Now we can refer to random as 'rand' instead
```

Figure 44: Example of importing a library with an alias

We can then call functions contained in an imported module by typing the name or alias of that module, followed by a dot (".") and the name of the function we are trying to call.

```
In [3]: for i in range (5):
            x = rand.randint(0, 5) # This returns a random integer between 0 and 5
            print(x)

4
2
3
3
5
```

Figure 45: Example of using a function from an imported module

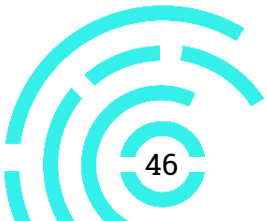
Useful modules for computational finance

There are several modules which provide functionality that is particularly useful for working with financial data. We will begin in this section by looking at NumPy.

NumPy

NumPy is a Python module which introduces a new data structure, the **NumPy array**, along with some operations to make working with these arrays easier. Compared to Python lists, which we covered in the previous section, NumPy arrays have a few differences:

> NumPy arrays contain only a single type of data, compared to Python lists, which can be made up of multiple distinct data types.



NumPy arrays allow for more efficient operations due to the concept of **vectorization**, which we will cover in detail in the next section. This makes NumPy arrays much more suitable when working with large datasets.

Assuming NumPy has been installed on your computer (this would have been installed as part of Anaconda if you followed the recommended installation guide for Jupyter Notebook), it is simple to import:

```
In [1]: import numpy # This gives us access to the numpy module
```

Figure 46: Example demonstrating how to import NumPy

We can create a NumPy array by calling the module's **array** function and passing in a list of values with which to initialize the array. This function returns a newly created NumPy array.

```
In [2]: numbers = numpy.array([1,2,3,4,5])
        print(numbers)

        [1 2 3 4 5]
```

Figure 47: Example of creating a NumPy array

We can also create matrices by calling the **array** function with a list of lists.

```
In [3]: matrix = numpy.array([[0, 1, 2], [3, 4, 5], [6, 7, 8]])
        print(matrix)

        [[0 1 2]
         [3 4 5]
         [6 7 8]]
```

Figure 48: Example of creating a matrix with the NumPy array

There are many useful functions for working with these arrays:

1 **len:** outputs the length of an inputted array.

```
In [4]: numbers = numpy.array([1,2,3,4,5])
        len(numbers)

Out[4]: 5
```

Figure 49: Example of the usage of the len function

2 **sum:** sums the elements of an array (assumes the array contains numeric elements).

```
In [5]: numbers = numpy.array([1,2,3,4,5])
        numbers.sum()

Out[5]: 15
```

Figure 50: Example of the usage of the sum function

3 **std:** calculates the standard deviation of elements in the array (assumes the array contains numeric elements).

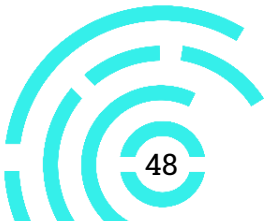
```
In [6]: numbers = numpy.array([1,2,3,4,5])
        numbers.std()

Out[6]: 1.4142135623730951
```

Figure 51: Example of the usage of the std function

4 **shape:** returns the dimensions of an array (technically this is not a function, but rather an attribute stored as part of the array data type, which is why we don't use rounded brackets when "calling" it).

```
In [7]: matrix = numpy.array([[0, 1, 2], [3, 4, 5], [6, 7, 8]])
        matrix.shape

Out[7]: (3, 3)
```

Figure 52: Example of accessing the shape attribute of a NumPy array



We can use the same square bracket notation of Python lists to access specific items in a NumPy array. We can also use this notation to access particular rows or columns of a matrix.

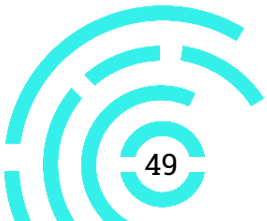
```
In [8]: numbers = numpy.array([1,2,3,4,5])
        print(numbers[2])

        3

In [9]: matrix = numpy.array([[0, 1, 2], [3, 4, 5], [6, 7, 8]])
        print(matrix[0,:]) # access the entire first row

        [0 1 2]

In [10]: matrix = numpy.array([[0, 1, 2], [3, 4, 5], [6, 7, 8]])
         print(matrix[:,1]) # access the entire second column

         [1 4 7]
```

Figure 53: Accessing items, rows and columns in NumPy arrays

Conclusion

In this set of notes, we covered conditional statements, took a deeper look at functions and learned how to declare our own functions. We also took an introductory look at importing external modules to expand Python's functionality. We then looked specifically at the functionality offered by the NumPy module, which is very commonly used for financial programming applications.

In the next section, we will look at three more crucial external modules – namely, Pandas, SciPy and Matplotlib, and learn how to import and use these. We will then go over the concept of vectorization in detail.



Unit 4 : External Modules and Vectorization

Unit 4 : Video Transcript

Welcome to the final video in the Introduction to Programming in Python module. In this video, we will cover a further three external modules – namely, Pandas, SciPy and Matplotlib – which are very useful for computational finance. We will then briefly discuss the concept of vectorization to conclude the module.

Let's begin by discussing Pandas. This is a module which introduces a further two data structures, **Series** and **DataFrames**. Pandas data structures use NumPy arrays underneath, and often we construct these data structures by passing in NumPy arrays. Pandas data structures are commonly used because they are more flexible and user-friendly than many of the other options available.

It is common practice to import pandas with the alias "pd", as demonstrated.

```
In [1]: import pandas as pd
```

Series are labelled one-dimensional data structures which can hold any data type. We refer to the axis labels collectively as the **index**. If no specific index is provided, it will automatically be generated to be the same as a normal array (numerically starting from zero).

```
In [2]: ages = pd.Series([21, 17, 18, 20])
        ages

Out[2]: 0    21
        1    17
        2    18
        3    20
        dtype: int64
```

However, we can provide more meaningful labels quite easily, by passing these in as a list of strings. Then, when we want to access the data relating to a particular label, we use the square bracket notation with that label instead of using a numerical index.

```
In [3]: ages = pd.Series([21, 17, 18, 20], ["Harry", "Joe", "Steven", "Josh"])
        ages
```

```
Out[3]: Harry     21
        Joe       17
        Steven    18
        Josh      20
        dtype: int64
```

```
In [4]: ages["Harry"]
```

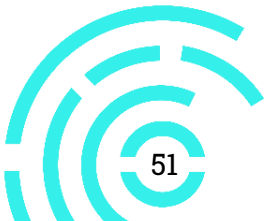
```
Out[4]: 21
```

DataFrames are labelled two-dimensional data structures wherein each column is made up of a common data type. These are essentially tables with column and row labels. DataFrames, like Series, have an index which labels the rows of the data structure, but DataFrames also add a second index to label the columns. This is referred to simply as **columns**. Similarly, to the case of Series, if we do not provide a specific index for rows or columns, they will simply be indexed from zero.

```
In [5]: import numpy as np

        prices = np.array([[10, 12, 14], [3, 4, 2], [1, 5, 7]])

        share_prices = pd.DataFrame(prices)
        share_prices
```

Out[5]:

	0	1	2
0	10	12	14
1	3	4	2
2	1	5	7



However, we can provide meaningful labels in order to make our data easier to work with and more readable to others.

In [6]:
```
prices = np.array([[10, 12, 14], [3, 4, 2], [1, 5, 7]])
names = ["Company A", "Company B", "Company C"]
dates = ["19/06/2018", "26/06/2018", "3/07/2018"]

share_prices = pd.DataFrame(prices, index=names, columns=dates)
share_prices
```

Out[6]:

	19/06/2018	26/06/2018	3/07/2018
Company A	10	12	14
Company B	3	4	2
Company C	1	5	7

In [7]:
```
share_prices["19/06/2018"]
```

Out[7]:
```
Company A    10
Company B     3
Company C     1
Name: 19/06/2018, dtype: int32
```

In [8]:
```
share_prices["19/06/2018"]["Company A"]
```

Out[8]:
```
10
```

We will now talk briefly about SciPy. This is a module which implements advanced routines for primarily scientific work. It enables Python to solve differential equations and optimization problems, amongst other tasks. SciPy is organized into many different sub-modules which each focus upon a different area of scientific computing.

As the course progresses, you will be introduced to the relevant SciPy submodules for each section.

Finally, let's now talk about Matplotlib. This is a module which implements data visualization tools into Python. This allows us to easily plot professional grade figures with the data that we are working with directly in Python, rather than having to export the data and plot with a different program.

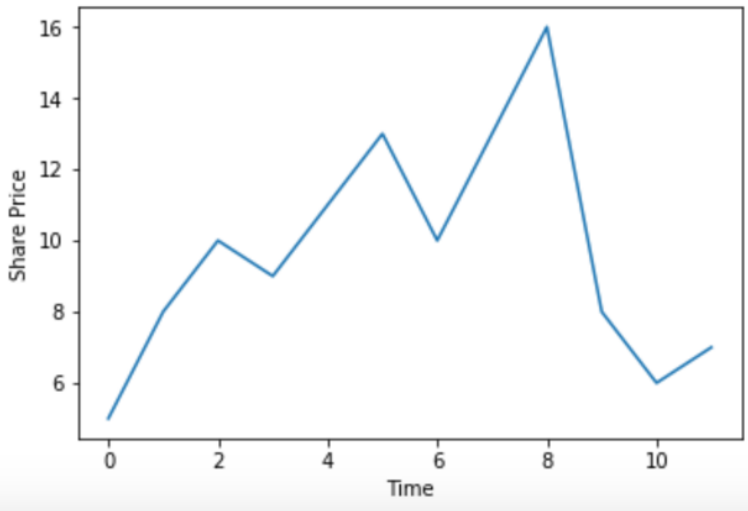
We tend to make extensive use of the pyplot sub-module of Matplotlib, so usually we will directly import this as shown:

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

We use the **xlabel** and **ylabel** functions to change the labels on each axis of our figures, and use the **plot** function, passing in the data we wish to graph, to create a simple line chart.

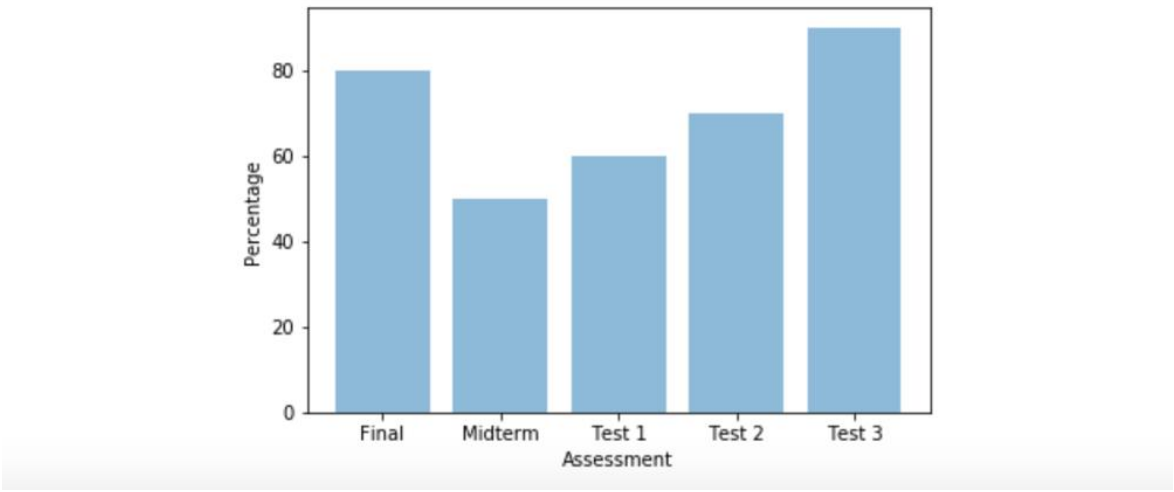
Importantly, we must use the **show** function to actually display our figure.

```
In [2]: import matplotlib.pyplot as plt

share_price=[5, 8, 10, 9, 11, 13, 10, 13, 16, 8, 6, 7]

plt.xlabel("Time")
plt.ylabel("Share Price")

plt.plot(share_price)

plt.show()
```



There are many other types of charts we can plot with Matplotlib, including bar charts, histograms and pie charts, amongst others.

```
In [3]: import matplotlib.pyplot as plt

        percentages=[60, 70, 50, 90, 80]
        tests = ["Test 1", "Test 2", "Midterm", "Test 3", "Final"]

        plt.xlabel("Assessment")
        plt.ylabel("Percentage")

        plt.bar(tests, percentages, alpha=0.5)

        plt.show()
```



We will now briefly discuss **vectorization**. This is the ability to perform an operation on an entire dataset at once, rather than having to individually perform the operation for each element. This allows for a much more efficient execution of code, which is very important when working when working with large datasets. This makes vectorization applicable to fields such as computational finance, where we often make use of very large sets of data.

When working with NumPy arrays, we can simply use an array in an expression and the output will be a new array constructed from the results of the expression with each array value substituted in. However, this is done much faster than if we were to use a loop to manually access each element and calculate the values individually.

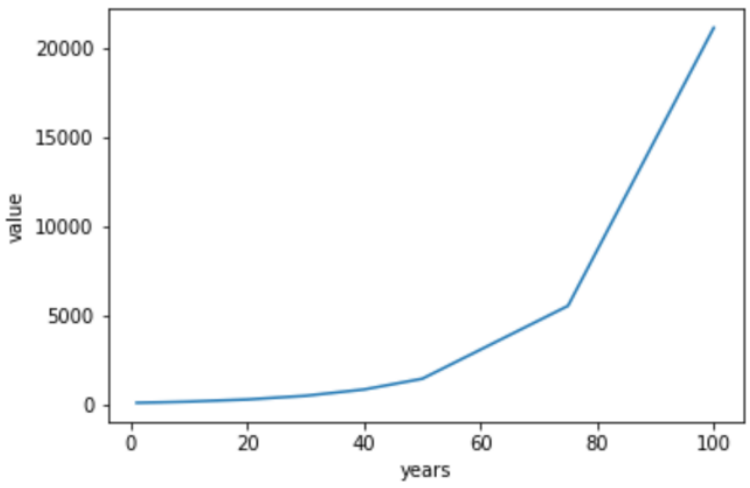
In [1]:
```
import numpy as np

IN = np.array([1,2,3,4,5])
OUT = IN ** 2

print(OUT)
```

```
[ 1  4  9 16 25]
```

In [2]:
```
principal = 100
interest_rate = 0.055 # per annum, compounded annually
years = np.array([1, 5, 10, 20, 30, 40, 50, 75, 100])

compounded = principal * ((1+interest_rate)**years)

print(compounded)
```

```
[   105.5          130.69600064    170.81444584    291.77574906
    498.39512884   851.3308774    1454.19612045   5545.42035913
  21146.86356738]
```

In [3]:
```
import matplotlib.pyplot as plt

plt.ylabel("value")
plt.xlabel("years")
plt.plot(years, compounded)
plt.show()
```



In this video, we covered three important modules – Pandas, SciPy and Matplotlib – and then discussed the increased efficiency provided by vectorization and its relevance to computational finance.

Unit 4: Notes

This set of notes offers a further exploration of useful external modules for computational finance, and a guide to the use of vectorization for working efficiently with large datasets.

Important modules for computational finance (cont.)

In the previous section, we covered the NumPy module and its array data structure. This module is used extensively in many other modules because of its efficient array data structure. We will now cover a further three modules – namely, Pandas, SciPy and Matplotlib – which are very useful for computational finance.

Pandas

Pandas is a module which introduces a further two data structures, **Series** and **DataFrames**. Pandas data structures use NumPy arrays underneath, and often we construct the Pandas data structures by passing in NumPy arrays. Pandas data structures are commonly used because they are more flexible and user-friendly than many of the other options available.

```
In [1]: import pandas as pd
```

Figure 54: Example of importing Pandas in Python

Series

Series are labelled one-dimensional data structures which can hold any data type. We refer to the axis labels collectively as the **index**. If no specific index is provided, it will automatically be generated to be the same as a normal array (numerically starting from zero).

```
In [2]: ages = pd.Series([21, 17, 18, 20])
        ages

Out[2]: 0    21
        1    17
        2    18
        3    20
        dtype: int64
```

Figure 55: Example of Series creation without a specified index

However, we can provide more meaningful labels quite easily, by passing these in as a list of strings. Then, when we want to access the data relating to a particular label, we use the square bracket notation with that label instead of using a numerical index.

```
In [3]: ages = pd.Series([21, 17, 18, 20], ["Harry", "Joe", "Steven", "Josh"])
        ages

Out[3]: Harry     21
        Joe       17
        Steven    18
        Josh      20
        dtype: int64

In [4]: ages["Harry"]

Out[4]: 21
```

Figure 56: Example of Series creation with a specific index

DataFrames

DataFrames are labelled two-dimensional data structures wherein each column is made up of a common data type. These are essentially tables with column and row labels. DataFrames, like Series, have an index which labels the rows of the data structure, but DataFrames also add a second index to label the columns. This is referred to simply as **columns**. Similarly, to the case of Series, if we do not provide a specific index for rows or columns, they will simply be indexed from zero.

```
In [5]: import numpy as np

        prices = np.array([[10, 12, 14], [3, 4, 2], [1, 5, 7]])

        share_prices = pd.DataFrame(prices)
        share_prices
```

Out[5]:

	0	1	2
0	10	12	14
1	3	4	2
2	1	5	7

Figure 57: Example of creating a DataFrame without specified row or column labels

However, we can provide meaningful labels in order to make our data easier to work with, and more readable to others.

```
In [6]: prices = np.array([[10, 12, 14], [3, 4, 2], [1, 5, 7]])
        names = ["Company A", "Company B", "Company C"]
        dates = ["19/06/2018", "26/06/2018", "3/07/2018"]

        share_prices = pd.DataFrame(prices, index=names, columns=dates)
        share_prices
```

Out[6]:

	19/06/2018	26/06/2018	3/07/2018
Company A	10	12	14
Company B	3	4	2
Company C	1	5	7

```
In [7]: share_prices["19/06/2018"]
```

```
Out[7]: Company A    10
        Company B     3
        Company C     1
        Name: 19/06/2018, dtype: int32
```

```
In [8]: share_prices["19/06/2018"]["Company A"]
```

```
Out[8]: 10
```

Figure 58: Example of creating a DataFrame with specified column and row labels

The Pandas documentation is available at this link: https://pandas.pydata.org/pandas-docs/stable/

SciPy

SciPy is a module which implements advanced routines for primarily scientific work. It enables Python to solve differential equations and optimization problems, amongst other tasks. SciPy is organized into many different sub-modules, each of which focuses on a different area of scientific computing. For example, the **linalg** sub-module deals with linear algebra problems, while the **stats** sub-module deals with statistical distributions and functions.

As the course progresses, you will be introduced to the relevant SciPy sub-modules for each section. The full documentation of the SciPy module is available at this link: https://jupyter.readthedocs.io/en/latest/running.html

Matplotlib

Matplotlib is a module which implements data visualization tools into Python. This allows us to easily plot professional grade figures with the data we are working with directly in Python, rather than having to export the data and plot with a different program.

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

Figure 59: Importing Matplotlib into a Python script

We use the **xlabel** and **ylabel** functions to change the labels on each axis of our figures, and use the **plot** function, passing in the data we wish to graph, to create a simple line chart. Importantly, we must use the **show** function to actually display our figure.

```
In [2]: import matplotlib.pyplot as plt

share_price=[5, 8, 10, 9, 11, 13, 10, 13, 16, 8, 6, 7]

plt.xlabel("Time")
plt.ylabel("Share Price")

plt.plot(share_price)

plt.show()
```

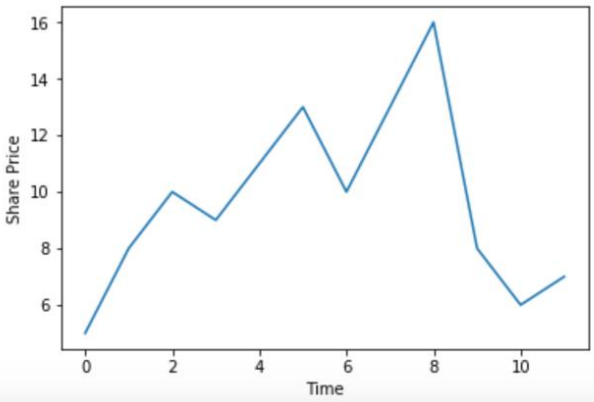


Figure 60: Example showing how to plot a simple line chart

There are many other types of charts we can plot with Matplotlib, including bar charts, histograms and pie charts, amongst others.

```
In [3]: import matplotlib.pyplot as plt

percentages=[60, 70, 50, 90, 80]
tests = ["Test 1", "Test 2", "Midterm", "Test 3", "Final"]

plt.xlabel("Assessment")
plt.ylabel("Percentage")

plt.bar(tests, percentages, alpha=0.5)

plt.show()
```

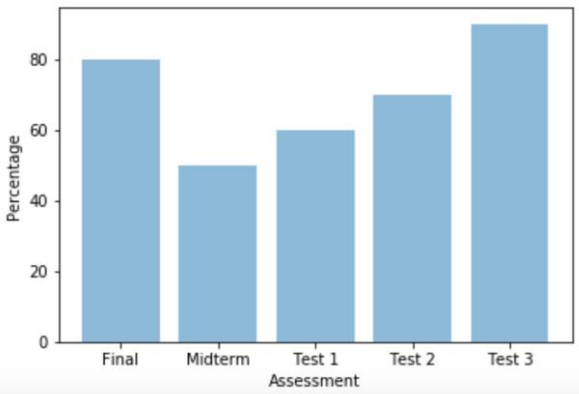


Figure 61: Example of plotting a bar chart

The full documentation of the module is available here if you would like to explore further: https://matplotlib.org/contents.html

Vectorization

Vectorization is the ability to perform an operation on an entire dataset at once, rather than having to individually perform the operation for each element. This allows for much more efficient execution of code, which is very important when working with large datasets. Therefore, vectorization is applicable to fields such as computational finance, where we often make use of very large sets of data. As was mentioned in the previous set of notes, this is one of the main reasons we tend to use NumPy arrays (or Pandas data structures) for working with large datasets.

When working with NumPy arrays, we can simply use the array itself in an expression, and the output will be a new array constructed from the results of the expression with each array value substituted in. However, this is done much faster than if we were to use a loop to manually access each element and calculate the values individually.

```
In [1]: import numpy as np

        IN = np.array([1,2,3,4,5])
        OUT = IN ** 2

        print(OUT)
```

```
[ 1  4  9 16 25]
```

Figure 62: Example of simple vectorization

```
In [2]: principal = 100
        interest_rate = 0.055 # per annum, compounded annually
        years = np.array([1, 5, 10, 20, 30, 40, 50, 75, 100])

        compounded = principal * ((1+interest_rate)**years)

        print(compounded)
```

```
[   105.5           130.69600064    170.81444584    291.77574906
    498.39512884    851.3308774    1454.19612045   5545.42035913
  21146.86356738]
```

```
In [3]: import matplotlib.pyplot as plt

        plt.ylabel("value")
        plt.xlabel("years")
        plt.plot(years, compounded)
        plt.show()
```

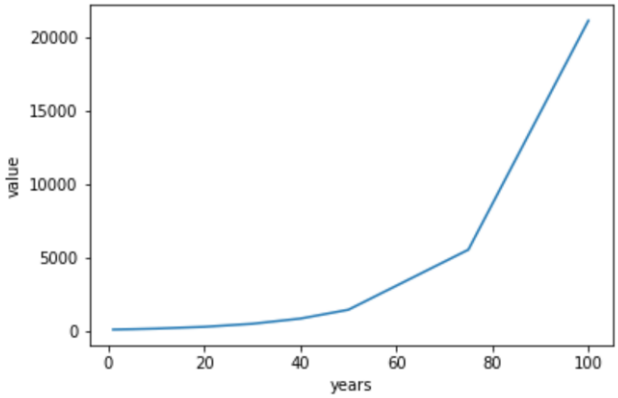


Figure 63: Example of slightly more advanced vectorization

Conclusion

In this section, we covered three important modules: Pandas, SciPy and Matplotlib. We explored their usage and syntax broadly, and indicated where one could go to learn more about each. Finally, we talked about the increased efficiency provided by vectorization and its relevance to computational finance.

This brings us to the end of Module 1: Introduction to Programming in Python. You should now have a firm grasp of the fundamentals of Python programming as well as some insight into the particular modules which are commonly used for computational finance. You are strongly encouraged to explore more of the features of these modules and Python itself.
To this end, there are many resources available online.

Bibliography

Heydt, M. (2015). *Mastering pandas for Finance*. Packt Publishing Ltd.

Hilpisch, Y. (2014). *Python for Finance: Analyze big financial data*. " O'Reilly Media, Inc.".

Hilpisch, Y. (2015). Derivatives analytics with Python: data analysis, models, simulation, calibration and hedging. John Wiley & Sons.

Idris, I. (2014). *Python data analysis*. Packt Publishing Ltd.

McKinney, W. (2012). Python for data analysis: Data wrangling with Pandas, NumPy, and IPython. " O'Reilly Media, Inc.".

Yan, Y. (2017). *Python for Finance*. Packt Publishing Ltd.

Collaborative Review Task

In this module, you are required to complete a collaborative review task that is designed to test your ability to apply and analyze the knowledge you have learned in the module.

Instructions

- Set up your workspace for this course by following **these instructions.**
- Work through the notebook linked in the Compiled Content M1 folder and the Collaborative Review Task submission page, and answer all questions/problems therein.
- You are allowed to consult the internet, as well as your peers on the forum.
- Your answers and solutions to the problems should be added to this notebook.
- Submit your final work as a .zip file.
- Note that Python (version 3.6.4) has been used to calculate the solutions.