THE PYTHON LANGUAGE FOR DATA SCIENCE

by Dr Juan H Klopper

- Research Fellow
- School for Data Science and Computational Thinking
- Stellenbosch University

INTRODUCTION

We begin some actual coding by exploring the basics of the Python programming language.

Python is a general purpose language. It can be used for creating standalone programs, games, websites, and much more.

Computer languages such as C and Julia are compiled languages. This means that before the code it executed it is converted completely into a machine language by a process termed compilation. This ensures fast execution. Python is instead an interpreted language. In Python each line of code is executed in line without compilation. Although slower, it makes for easier to understand code.

There are versions of Python. A current newest version is continuously being released. A base of core developers are tasked with these updates. Version 3 of the language has been availaible for some time. Currently version 3.10 is getting ready for release. All older version 2 releases have been deprecated and no development is taking place for these versions.

THE PYTHON ECOSYSTEM FOR DATA SCIENCE

Due to its exstensible nature and its ease of use, it has become the favourite programming language for Data Science. As a language for Data Science it can be scripted. This means that we can write short snippets of code (a line or a few lines of code) and explore the results, i.e. we do not have to write a whole program before executing it in order to get results. Iterating over this process of writing snippets allows for proper data analysis.

Extensibility refers to packages that have been developed that extend the core language. These packages are developed by thousands of individuals and groups from across the globe. There are packages that have been exclsusively designed for data managament and analysis. These packages are sometimes referred to as the scientific stack. In this course we learn about the most common packages in the scientific stack. This ensures that anyone taking this course will be well equipped to take on real-world problems.

The most common packages used in Data Science are numpy (short for numerical computing), scipy (short for scientific python) which extends the functionality of numpy. Matplotlib aand plotly are commonly used plotting libraries. We will use plotly is this course. It produces interactive plots that enhance our ability to understand data. Pandas is a data package that allows us to import and manipulate data. Scikit-learn is an excellent machine learning package. Lastly, we have to mention TensorFlow, the opne source package by Google for deep learning. Google has recently added the DecisionForest module to TensorFlow and these allow us to build random forest machine learning models. We will use this module to create these machine learning models.

TOOLS FOR USING PYTHON

We have learned about coding environments in the first notebook. These are the sotware programs that we use to type our Python code into. Commonly used coding environments include Jupyter notebooks, PyCharm, Spyder, and Microsoft Visual Studio Code.

Jupyter notebooks are run in an internet browser. One such example is Google Colab that uses a type of Jupyter notebook. Google Colab is our coding environment for this course. It is available as an app in Google Drive. Anyone with a Google email account has Google Drive and hence access to Google Colab. Using Google Colab means that we do not have to install any Python software or coding environment to our local computers. We can also easily share our work or communicate our results, the same as any Google app.

Remember to go to https://colab.research.google.com once you have signed up for a free Google account. This will add Google Colab to all the other Google Drive apps that are available to you.

It's time to start exploring the Python language. The simplest way to start is through simple arithmetic. Python can act as a giant calculator.

ARITHMETIC

We are all familiar with the basic arithmetical operators such as addition and multiplication. The simple use of mathematical operations in Python are examples of **expressions**. The actual symbols such as `+` and `-` are termed **operators**. TABLE 1 below shows a list of these operators.

TABLE 1

Expression	Operator	Example	Result
Adding	`+`	`2 + 2`	`4`
Subtracting	`-`	`8 - 3`	`5`
Multiplying	`*`	`2 * 2`	`4`
Dividing	`/`	`8 / 4`	`2`
Integer devision	`//`	`10 // 3`	`3`
The remainder	`%`	`10 % 3`	`1`
Exponentiation	`**`	`2 ** 4`	`8`

We explore these common expression below.

ADDITION

We start by adding $2 + 2$, which should results in 4. To excute a cell, we can hold down the SHIFT key on our keyboard and then hit the NETER key on a Windows or Linux machine or ENTER on a Mac. There is also a button the the left of ease cell in Google Colab that we could click.

```
1 2 + 2  # Hold SHIFT and hit ENTER (PC / Linux) or RETURN (Mac) to execute a cell
```

```
4
```

```
1 2 + 2
```

```
4
```

Note use of spaces between the 2 and the $+$ symbol. This is simply for easy of reading. We could also omit the spaces and write `2+2` . Note also the use of a code comment. A code

comment is started by a pound symbol or hashtag. Python ignores any script following a pound symbol (for a specific line of code). We use code comments to leave messages to ourselves and those that may use our code.

More than two numbers can be added in a single expression. Below we add 2 and 2 and 10.

```
2 + 2 + 10
```

```
14
```

SUBTRACTION

One number is subtracted from another using the `-` operator.

```
7 - 4
```

```
3
```

More than one number can be used in a subtraction expression.

```
10 - 3 - 4
```

```
3
```

MULTIPLICATION

Since keyboards do not have a multiplication key, we use the `*` symbol for this operation. It is ususally a part of the `8` key.

```
3 * 4  # Multiplication is the * symbol (SHIFT and 8)
```

```
12
```

```
3 * 4 * 2 # More than two numbers
```

```
24
```

DIVISION

As with multiplication, we make use of another key for this operation. It is the forward slash key, `/`.

```
1 10 / 2  # Division is the / key
```

```
5.0
```

Note that we used integers, but the results is expressed in decimal format.

```
1 20 / 8
```

```
2.5
```

The result of dividing 20 by 8 results in a value with a decimal point. We can use the `//` operator to return only the whole number (integer) part of the solution.

```
1 20 // 8  # Integer devision
```

```
2
```

Since $2 \times 8 = 16$, we have a remainder of 4 (to get to 20). We can express the remainder using the `%` operator.

```
1 20 % 8  # Remainder
```

```
4
```

POWERS

Consider the exponentiation below in (1).

$$2^3 = 2 \times 2 \times 2 = 8 \tag{1}$$

We use the double asterisk symbols, `**`, for taking powers.

```
1 2**3  # Power is a double asterisk ** symbol
```

```
8
```

THE ORDER OF ARITHMETICAL OPERATIONS

Remember that there is an order to mathematical operations, i.e. division and multiplication comes before subtraction and addition. In the expression $3 + 4 \times 2$, the 4 and 2 and multiplied first resulting in 8. The 2 is then added to yield 11.

```
1 3 + 4 * 2  # Multiplication before addition
```

```
11
```

Parentheses are used to force the order of operations. Below, we add $3 + 4$ first and then multiply the results by 2.

```
1 (3 + 4) * 2  # Force the order by using parentheses
```

```
14
```

Exercise

> Calculate the following

$$\frac{(6 + 4) \times 2}{10}$$

```
1 # Place your solution here
```

Solution

[ ] ↳ *1 cell hidden*

COMPARISON OPERATORS

Comparison operators are used to return the value of a comparison. The return is one of two value: `True` or `False` based on the comparison being made. The operators are listed in TABLE 2 below.

TABLE 2

Comparison	Operator	True example	False Example
Less than	<	2 < 4	4 < 2
Greater than	>	4 > 2	2 > 4
Less than or equal to	<=	4 <= 4	2 <= 4
Greater than or equal to	>=	4 >= 4	2 >= 4
Is equal to	==	4 == 4.0	4 == 2
Is not equal to	!=	4 != 2	4 != 4

Below, we run through the examples in TABLE 2. Follow the code comments.

```
1 # Less than
2 2 < 4
```

```
2 2 < 4
```

```
True
```

```
1 # Greater than
2 4 > 2
```

```
True
```

```
1 # Less than or equal to
2 4 <= 4
```

```
True
```

```
1 # Greater than or equal to
2 4 >= 4
```

```
True
```

```
1 # Value equality
2 4 == 4.0 # Although the type differs the values are equal
```

```
True
```

```
1 # Value inequality
2 4 != 2
```

```
True
```

```
1 # Less than
2 4 < 2
```

```
False
```

```
1 # Greater than
2 2 > 4
```

```
False
```

```
1 # Less than or equal to
2 4 <= 2
```

```
False
```

```
1 # Greater than or equal to
2 2 >= 4
```

```
False
```

```
1 # Value equality
2 4 == 2
```

```
2 4 == 2
```

```
False
```

```
1 # Value inequality
2 4 != 4
```

```
False
```

FUNCTIONS

A lot of what we do in Python requires a function. A **function** is a keyword in the language that takes some input (always provided inside of a set of parenthesis that directly follow the function keyword) and gives an output. An input is called an **argument** (or sometimes a parameter). Below, we pass the argument `'This is easy,'` to the function `print()`. The output is a screen printout of the input.

```
1 print('This is easy.')
```

```
This is easy.
```

As we continue, we will learn more and more functions. The functions or keywords in Python have rules of use. This is much like a spoken language. In essence, we are learning a new language. Not to worry, it is much simpler than learning a new spoken language.

DATA TYPES

Much of what we work with in Python are of a certain computer data type. This type sets the rules for what we can do.

One helpful function is the `type()` function. It tells us the computer data type that we are working with. Think of the number 3. In mathematics, it is an integer.

```
1 # Integer specified by the abbreviation int
2 type(3)
```

```
int
```

Once we add a decimal place, we change the computer data type. For instance 3.0 is a decimal value. These are termed floating point values in most computer languages.

```
1 # Floating point number specified by the abbreviation float
```

```
2 type(3.0)
```

```
float
```

Characters and text are termed strings. They are placed in single (or double) quotation marks.

```
1 # Text specified by the abbreviation str
2 type('KRG')
```

```
str
```

When numbers are placed in quotation marks, they become strings and we can no longer use them for calculations.

```
1 type('8')
```

```
str
```

Adding strings concatenate them. Below we add three strings to each other. Note the inclusion of spaces in each string.

```
1 'Python ' + 'is ' + 'a ' + 'powerful ' + 'language' + '.'
```

```
'Python is a powerful language.'
```

The `str` function converts its argument to a string. Below we use it to first calculate a solution and then convert it to a string.

```
1 'The solution to 1 + 1 is ' + str(1 + 1) + '.'
```

```
'The solution to 1 + 1 is 2.'
```

All the examples above are referred to as objects. The number 3 is an object and the string `Python` is an object.

THE MATH MODULE

One of the major strengths of Python is the community that grows the language. In many cases these are interested individuals who write packages and modules that extend the core language with new functions. These are made freely available and we can import them. This is done with the `import` command. The math package is actually an in built Python package and provides access to the mathematical functions defined by the C language standard.

Note: If you are using Python on a local computer, packages that are not built in Python packages have to be installed prior to importing them.

```
1 # Import the math module
2 import math
```

This module adds many mathematical constants and functions when imported. Below we take a look at a few. You can learn more at https://docs.python.org/3/library/math.html

Once a package has been imported, we can now use its functionality. To use the function in a package we have to refer to the package name and use dot notation. We start my looking at the built in constants.

MATH CONSTANTS

Two of the most famous irrational numbers are π and e (Euler's number). To use these constants, we need to specify the name of the package followed by a dot and then the constant name.

```
1 math.pi # Note the use of dot notation
```

```
3.141592653589793
```

```
1 math.e
```

```
2.718281828459045
```

The `math.e` constant is the same as the `exp` function in the math module. We pass an argument of `1` to this function, to replicate what we would do in mathematics, where every number raised to the power 1, is just that number. This is shown in (2).

$$e^1 = e \tag{2}$$

```
1 math.exp(1)
```

```
2.718281828459045
```

TRIGONOMETRIC FUNCTIONS

The math package provides for all the trigonometric function. We use keywords such as `math.sin` for the sine function and `math.tan` for the tangent function. The argument that we pass must be in radians. In (3) we see a sine function and then we replicate it in code

$$\sin\left(\frac{\pi}{2}\right) = 1 \tag{3}$$

```
# Sine
math.sin(math.pi / 2)
```

```
1.0
```

In (4) we have a cosine function.

$$\cos\left(\frac{\pi}{2}\right) = 0 \tag{4}$$

```
# Cosine
math.cos(math.pi / 2)
```

```
6.123233995736766e-17
```

We note a result with 16 zeros after the decimal point. This is a truncation error and the result is, in essence, 0.

Exercise

> Calculate the following

$$\tan\left(\frac{\pi}{4}\right)$$

```
# Write your solution here
```

Solution

[ ] ↳ *1 cell hidden*

LOGARITHMS

Consider the function in (5).

$$\log_a b = c \tag{5}$$

Since 1000 is 10 to the power 3, $\log_{10} 1000 = 3$. The `math.log10` function calculates $\log$ based 10.

```
# Base 10 logarithm
math.log10(1000)
```

```
3.0
```

The `math.log` function has a base of e (Euler's number). This calculate the natural logarithm.

```
# Natural logarithm
math.log(math.e)
```

```
1.0
```

ROOTS

The `math.sqrt` function calculates the square root, shown by example in (6).

$$\sqrt{256} = 16^2 \tag{6}$$

```
math.sqrt(256)
```

```
16.0
```

IMPORT ALTERNATIVES

There are alterantives to importing the functionality from a package. One such alternative is the use of a namespace abbreviation. Namespace refers to the functionality in a package (all the names of the function in a package for instance). We could then use the abbreviation below.

```
import math as m
```

To use a function such as `sqrt` we can now type `m.sqrt` instead of `math.sqrt`.

```
# Square root of 9
m.sqrt(9)
```

```
3.0
```

We can also import a function or functions by name.

```
# Importing the sine and cosine functions and the pi constant
from math import sin, cos, pi # Note the sepaartion by a comma
```

Now we don't have to use the `math` or `m` dot notation.

```
# Sine of pi over 2
sin(pi / 2)
```

```
1.0
```

If we want to import all the functions in a package we can use the `*` wildcard.

```
from math import *
```

```
# No dot notation
log10(100)
```

```
2.0
```

Now all the functions and constants can be used without dot notation. We need to caution against this approach through. Some packages have function names that are the same as Python core functions. These will then overwrite the core function, making them inaccessible.

Most packages have subpackages, sometimes called modules. For instance the scipy package has a stats module containing many functions for statistical analysis. We will learn how to import and use these modules during teh course.

Although any abbreviation can be used, some have become very common such as `np` for numpy. In this course we will use these common abbreviations.

COMPUTER VARIABLES

A **computer variables** is a name that we give to a space in computer memory where we want to save an object to. An object is a piece of information with a type. Remember that `3` is of type `int`. Objects include simple numbers, strings, and the solution to calculations. We have seen the `type` function that gives us the type of an object.

There are some restrictions and suggestions to the names we can use for computer variables.

- Use names that start with a lowercase letter
- Make it descriptive

- Use snake case `my_variable_name` (or the less popular camel case `myNewVariable`)
- Don't use `python` functions or keywords

CREATING AN OBJECT AND ASSIGNING IT TO A COMPUTER VARIABLE

Below, we assign an object of type `float` to a computer variable that we choose to be `my_value`. The equal symbol, `=`, is the assigment operator in Python, and is not the familiar equal sign in mathematical functions.

```
# Create a computer variable to hold a floating point object
my_value = 3.0  # Assign an object which is a  floating point number to a comput
```

We can recall the object that in now saved in memory by calling the computer variable name. Calling a variable to display or use its conent is done by simply typing it.

```
# Recall the object in the computer variable
my_value
```

```
3.0
```

```
# Type of object held in the computer variable
type(my_value)
```

```
float
```

The *content* in the computer variable can be updated with new values or objects.

```
# Starting value
i = 0
i
```

```
0
```

The computer variable `i` now holds an integer, the value being 0. Below we add 1 to the already held value.

```
# Add 1 to i
i = i + 1
i
```

```
1
```

The assigment operator assigns what is to its right to what is to its left. So, in the code above, `i` already held the value 0. In essence our code read `i = 0 + 1` and now `i=1`.

It is typical to use shorthand notation as seen below.

```
# Short-hand update
i += 1
i
```

```
2
```

Exercise

> Do the following:

- Create a computer variable to hold the string 'This is easy.'
- Recall the computer variable
- Confirm that the computer variable indeed holds an object that is a string

```
# Create your computer variable for assigment here
```

```
# Recall the variable
```

```
# Determine the type of the object assigned to the computer variable
```

Solution

[ ] ↳ *4 cells hidden*

COLLECTIONS

It becomes useful to work with more than just a single value. For this, Python has a variety of objects that can contain a collection of elements. We start with list objects.

LIST OBJECTS

A list in Python is created using a set of square brackets. The elements are all seprarated by a comma.

```
# A list object conatining only a single element
[10]
```

```
[10]
```

```
1 # Create a list with five elements using square brackets and a comma between eac
2 [1, 2, 3, 4, 5]
```

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

For the sake of brevity we can omit the term object. Here then, we will simply stat a *list* instead of a *list object*.

Lists can have objects of different data types.

```
1 # List with elemnts of various types
2 [1, 'two', 3.]
```

```
[1, 'two', 3.0]
```

The `type` function will return a `list` data type.

```
1 # Type of the object
2 type([1, 'two', 3.])
```

```
list
```

As before, we can assign a list object to a computer variable name.

```
1 # Creating a list and saving it in a computer variable
2 my_list = [1, 2, 3, 4]
```

```
1 # Recall the content of the computer variable
2 my_list
```

```
[1, 2, 3, 4]
```

When we pass a list as argument to the `len` function, it returns the number of elements in the list.

```
1 len(my_list)  # Using the len() function
```

```
4
```

In Python, objects have methods and attributes (properties). We use these by placing a dot `.` after the object and then writing the method or the attribute name. **Methods** are like functions and **attributes** (properties) just tell us something about the object. Because methods are like functions, we end them with parenthesis and we can pass arguments to them. Attributes have no parenthesis.

The `count` method takes an argument value that it then *looks for* in the object. It returns how many times the argument appears in an element of the list.

```
# Using the .count method and looking at how many times 3 occurs in the list ob
my_list.count(3)
```

```
1
```

We can create lists inside of lists.

```
# Creating a nested list object
my_nested_list = [[1, 2, 3], [3, 4, 5]]
my_nested_list
```

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

The `len` function counts the number of these nested lists.

```
# Length of the nested list
len(my_nested_list)
```

```
2
```

The `append` method can add new elements to the end of a list.

```
# Add the integer 5 to my_list
my_list.append(5)
```

```
# View the appended list object
my_list
```

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

The `pop` method removes the last element and displays that element to the screen.

```
# Remove the last element
my_list.pop()  # Will show the removed element
```

```
5
```

```
# View the list object
my_list
```

```
[1, 2, 3, 4]
```

Every element in a list has a *adress* called its **index**. We can request the value held at an index or even request more than one value. Below, we create a new list named `my_other_list` which contains three elements.

```
# Create my_other_list
my_other_list = [10, 20, 30]
my_other_list
```

```
[10, 20, 30]
```

Python indexing starts at 0. That means the index of the first elemenst is 0. If we have n elements in a list, its last element will have an index of $n - 1$.

```
# Show first element in my_other_list
my_other_list[0]  # Python is zero-indexed
```

```
10
```

```
# Show second element
my_other_list[1]
```

```
20
```

```
# Third and last element
my_other_list[2]
```

```
30
```

The index `-1` retruns the last element. It is convenient if we do not know how many elemenst are in a list.

```
# Short hand for showing the last element
my_other_list[-1]
```

```
30
```

A **slice** of the list can return more than one element. Below we use a colon, `0:2`, in our indexing. It will return objects zero and one. When using indexing in this way, the last element (`2` here) is not included.

```
# Show first and second elements
my_other_list[0:2]  # Last index-value is not included
```

```
[10, 20]
```

Nested lists can make use of more than one index. Below, we recall `my_nested_list`.

```
# Recall my_nested_list
my_nested_list
```

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

The second element (index 1) in the first nested list (index 0) is returned below.

```
# Show second element of the first nested list of my_nested_list
my_nested_list[1][0]
```

```
3
```

Indexing can be used to change the content of a list. Below, we change the last element in `my_other_list` to -40.

```
# Change the last element in my_other_list to -40
my_other_list[-1] = -40
my_other_list
```

```
[10, 20, -40]
```

Double colon symbols, `::` can be used to include all element from the start up to a certain value or from a ceratin value to the end of the list.

```
my_other_list[1::]  # Element 2 until the end
```

```
[20, -40]
```

Exercise

> Do the following

- Create a list with 10 elements from 0 through 9
- Use indexing to print the first 5 elements of the list
- Use indexing to print every second element (Hint: The colon symbol can also include a step size, i.e. `0:10:2` would be elements `0, 2, 4, 6, 8`, as the step size is set to `2`)

```
# Create the list here (with an appropriate name)
```

```
# Print the first five elements
```

```
# Everey second element
```

Solutions

[ ] ↳ *3 cells hidden*

TUPLES

Tuples are much like lists. The elements are immutable, though. This means that they cannot be changed once created.

```
1 my_tuple = (3, 5, 'two')
2 my_tuple
```

```
(3, 5, 'two')
```

```
1 my_tuple[-1] # Each element still has an index
```

```
'two'
```

Elements of a tuple can be named over an above having an index. Below we assign the computer variable names `one`, `two`, and `three` to each element of `my_tuple`.

```
1 one, two, three = my_tuple
```

We can now recall a value by using its name.

```
1 one # Recalling one
```

```
3
```

DICTIONARIES

Dictionaries are collections where each element is a key-value pair. Every value has a key (a name). Python uses curly braces to indicate dictionaries.

```
1 # create a dictionary of key-value pairs
2 my_dict = {'Language':'Python',
3            'Version':3,
4            'Environment':'Colab'}
```

We can call up the keys with the `keys` method and the values with the `values` method.

```
# Keys
my_dict.keys()
```

```
dict_keys(['Language', 'Version', 'Environment'])
```

```
# Values
my_dict.values()
```

```
dict_values(['Python', 3, 'Colab'])
```

All the key-value pairs can be retrieved using the `items` method.

```
# Retrieve keys and values
my_dict.items()
```

```
dict_items([('Language', 'Python'), ('Version', 3), ('Environment', 'Colab')])
```

To return the value of a specific key, we use the `get` method.

```
# Value of Version key
my_dict.get('Version')
```

```
3
```

Instead of curly braces, the alternative syntax uses a list of tuples as a list and the latter as argument to the `dict` function.

```
# Alternative syntax with list of tuples
my_other_dict = dict([('Language', 'Python'),
                      ('Version', 3.8),
                      ('Environment', ['Spyder', 'Notebook'])])
my_other_dict
```

```
{'Environment': ['Spyder', 'Notebook'], 'Language': 'Python', 'Version': 3.8}
```

LOOPS

Loops allows us to run over some code many times, depending on some criteria. Here, we investigate the `for` and the `while` loops.

FOR LOOP

The `range` function is convenient for generating sequences. With a single-value integer argument it generates a list starting at 0 and increments in a step size of 1 until the specified value minus 1. The specified value is not included. The function returns a range object that can be used for iteration (over its elements).

```
range(10) # Generating the values 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
```

```
range(0, 10)
```

Below, we see an example of a `for` loop. We generate a loop counter, `i` which will loop over the values specified in the `range` function. Each of the values in the range is printed during the iteration.

```
# Loop through 0 to 9
for i in range(10):
  print(i)
```

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

We can loop through the elements of a list object too. The `element` name down below is just a placeholder name.

```
my_other_list # Recall the object
```

```
[10, 20, -40]
```

```
# Loop through elemnts in my_other_list and print them
for element in my_other_list:
  print(element)
```

```
10
20
-40
```

Below, we use a `for` loop to print the keys of a dictionary.

```
# Loop through keys in my_dict and print them
for key in my_other_dict.keys():
  print(key)
```

```
Language
```

```
Version
Environment
```

WHILE LOOP

We can also loop over code while a condition is met. We start with a counter and then use the `while` loop which will continue until a condition is met, or more precisely, the loop terminates when the condition returns a `False` value.

```
# Print values while less than 5
i = 0  # Start value
while i < 5:
  print(i)
  i += 1 # Increment by 1
```

```
0
1
2
3
4
```

Remember that `i < 5` is a conditional and will return either a `True` or a `False` value. The loop will continue as long as a `True` value is returned.

Exercise

> Loop over the following

- Create a computer variable called `my_string` and assign the string object 'This is easy.` to it
- Loop over each character in the string and print it to the screen

```
# Generate the list object
```

```
# Calculate the number of characters
```

```
# Loop over each character and print it to the screen
```

Solution

[ ] ↳ *3 cells hidden*

USING CONDITIONAL OPERATORS

Sometimes, we only want to execute code when some condition is met. The `if`, `elif`, and `else` statements all help us do this.

Below we store an integer value in a computer variable. When then set a condition. If the integer object's value is less than a certain number, we print something to the screen, but if it is not, then we print something else to the screen.

```
# Check on condition
i = 23 # Set the value of the integer

if i < 20:
  print('Smaller than 20')
else:
  print('Equal to or greater than 20')
```

```
Equal to or greater than 20
```

Since $23 > 20$ the second string prints to the screen.

We can concatenate more than one condition by inserting the `elif` keyword.

```
# Using elif
a = 2
b = 1
if b > a:
  print("b is greater than a")
elif a == b:
  print("a and b are equal")
else:
  print("a is greater than b")
```

```
a is greater than b
```

Example excerise

In this famous computer task, we take a range of numbers (0 through 21 in this case). For each number, if it is divisible by some integer, say 3 (no remainder), then we print `foo`, else we print `bar`.

```
# foo-bar
for i in range(22):
  if i % 3 == 0:
    print(i, ' : foo')
  else:
    print(i, ' : bar')
```

```
0  : foo
1  : bar
2  : bar
3  : foo
4  : bar
5  : bar
6  : foo
7  : bar
8  : bar
9  : foo
10  : bar
11  : bar
12  : foo
13  : bar
14  : bar
15  : foo
16  : bar
17  : bar
18  : foo
19  : bar
20  : bar
21  : foo
```

LIST COMPREHENSION

At times it is convenient to generate a list using a loop with or without conditionals. This is a more advanced topic and is only introduced here.

Below, we geberate a list containing the integers 0 through 50.

```
1 my_set = [i for i in range(51)]
2 my_set[0:5]  # Print first 5
```

```
[0, 1, 2, 3, 4]
```

Now we show every number in the list that is divisible by 3.

```
1 [i for i in my_set if i % 3 == 0]
```

```
[0, 3, 6, 9, 12, 15, 18, 21, 24, 27, 30, 33, 36, 39, 42, 45, 48]
```

As a more complicated example, we square each value in the list if that value is divisible by 5.

```
1 [i**2 for i in my_set if i % 5 == 0]
```

```
[0, 25, 100, 225, 400, 625, 900, 1225, 1600, 2025, 2500]
```

THE NUMPY LIBRARY

At the root of many computations and also as prerequisite for many packages in Python is the numerical Python package numpy. It is actually written in C wrapped into Python functions that execute the advanced code much faster than core Python. We examine some of the useful functions in the numpy package below. Many of them replace the functionality that we have seen in this notebook.

ARRAYS

Arrays are much like Python lists. Before we can use numpy, we have to import it. We do so with a namespace abbreviation, `np`, that is commonly used.

```
1 import numpy as np
```

We generate arrays with the `array` function. The elements are still passed as a list.

```
1 my_array = np.array([0, 1, 2, 3]) # Note the use of the namespace abbreviation
2 my_array
```

```
array([0, 1, 2, 3])
```

```
1 my_other_array = np.array(range(10)) # Values 0 through 9
2 my_other_array
```

```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

Arrays have very useful methods. Some of these methods are also available as numpy functions.

ARITHMETIC ON NUMPY ARRAYS

Below we list some of the useful arithmetical operations we can use with arrays. They are specified as code comments.

```
1 # Adding the elements of an array
2 my_other_array.sum()
```

```
45
```

Since the mean (or average) of a set of values is simply their sum divided by their count (which we can calculate using the `len` function), we can easily calculate the mean of the values in

```
my_other_array.
```

```
1 # Mean of values in array
2 my_other_array.sum() / len(my_other_array)
```

```
4.5
```

There is an easier way to achieve this by using the `mean` method.

```
1 my_other_array.mean()
```

```
4.5
```

CONCLUSION

This was a brief, but as we shall see later, a very useful introduction to Python and using scripts in the cells of a notebook. Python is a well-structured, easy to understand language that we can use in scripting form to do Data Science.

```
1
```