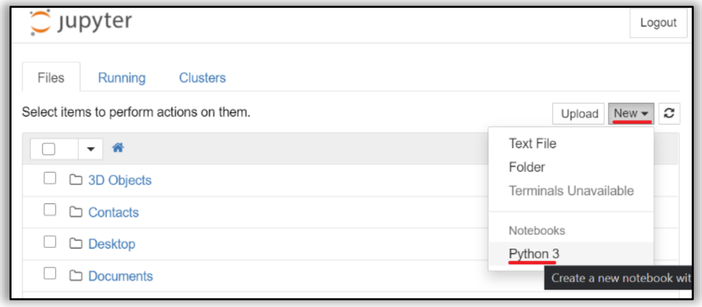
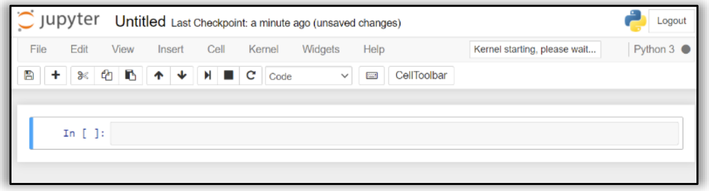
BRIEF INTRODUCTION TO PYTHON

In this week we will go through a number of basic constructs of Python programming language. Once you complete the tutorial you will have the knowledge of confidently putting together various programming block and structures to successfully develop a program in Python.

Step 1

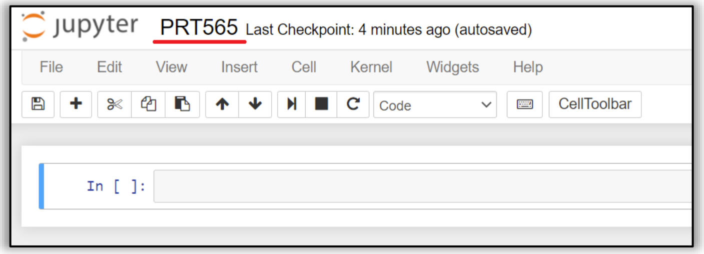
Open up Jupyter Notebook using steps shown in previous tutorial, now bring up a Python 3 Script as shown below:

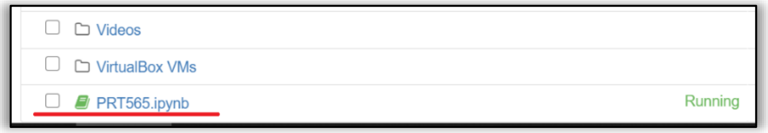



jupyter
Untitled Last Checkpoint: a minute ago (unsaved changes)
Logout
File
Edit
View
Insert
Cell
Kernel
Widgets
Help
Kernel starting, please wait...
Python 3
Code
CellToolbar
In [ ]:

Step 2

Now rename the unsaved notebook as PRT565 (From *File* > *Rename*). You should see that the filename has changed. Now this file will be available even in later stage from the list of files shown upon opening the Jupyter Notebook (normally at the bottom).

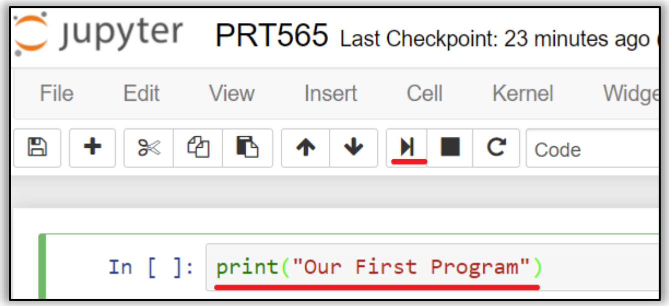




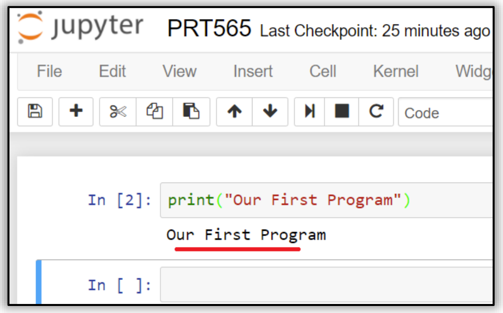
Running basic python codes with Google Colab

Construct 1 – Displaying Texts

Our First Program will simply output or display a line of text in the screen. To do that we will use the ***print()*** function. Type out the string "Our First Program" as shown below and execute the cell using the execute button and press *Shift+Enter*. You will see the string being displayed.



```
In [ ]: print("Our First Program")
```



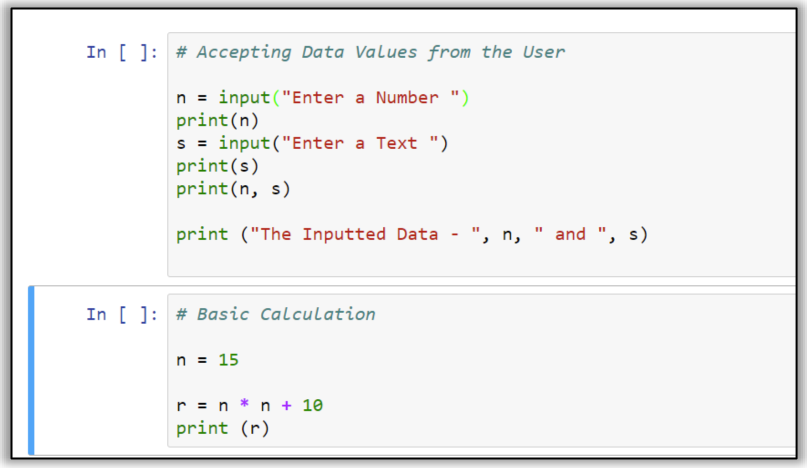
```
In [2]: print("Our First Program")
        Our First Program

In [ ]:
```

Construct 2 – Accepting Input from User

Now for the next program you can either chose to go with another cell or you can clear the output of the current cell through *Cell > All Output > Clear*.

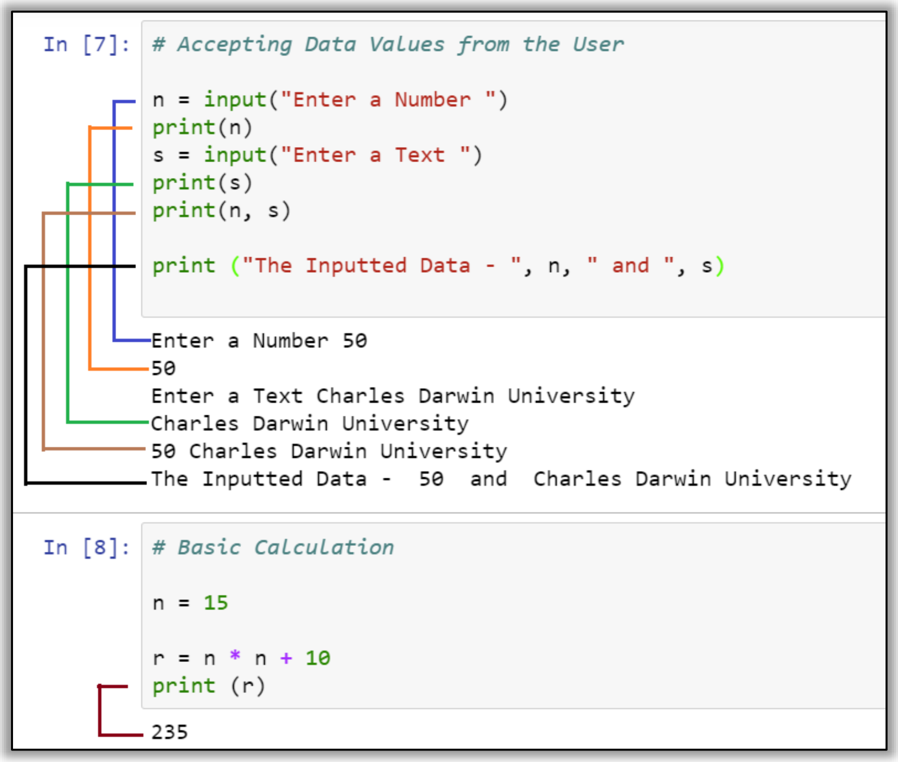
In this program we will use the ***input()*** function to accept numbers and texts from the user into *Variables*. Variables are placeholders in computer memory where you can save different values during the execution of the program. The variable used in this program are ***n*** and ***s***.

Here the program is broken down into cells. You can execute all the cells at a time through *Cell > Run All*. In addition, note that while writing comments in python, the hashtag (#) is used.



```
In [ ]: # Accepting Data Values from the User

        n = input("Enter a Number ")
        print(n)
        s = input("Enter a Text ")
        print(s)
        print(n, s)

        print ("The Inputted Data - ", n, " and ", s)

In [ ]: # Basic Calculation

        n = 15

        r = n * n + 10
        print (r)
```

Once the program is executed you should get the similar output as below:



```
In [7]: # Accepting Data Values from the User

        n = input("Enter a Number ")
        print(n)
        s = input("Enter a Text ")
        print(s)
        print(n, s)

        print ("The Inputted Data - ", n, " and ", s)

        Enter a Number 50
        50
        Enter a Text Charles Darwin University
        Charles Darwin University
        50 Charles Darwin University
        The Inputted Data -  50  and  Charles Darwin University

In [8]: # Basic Calculation

        n = 15

        r = n * n + 10
        print (r)

        235
```

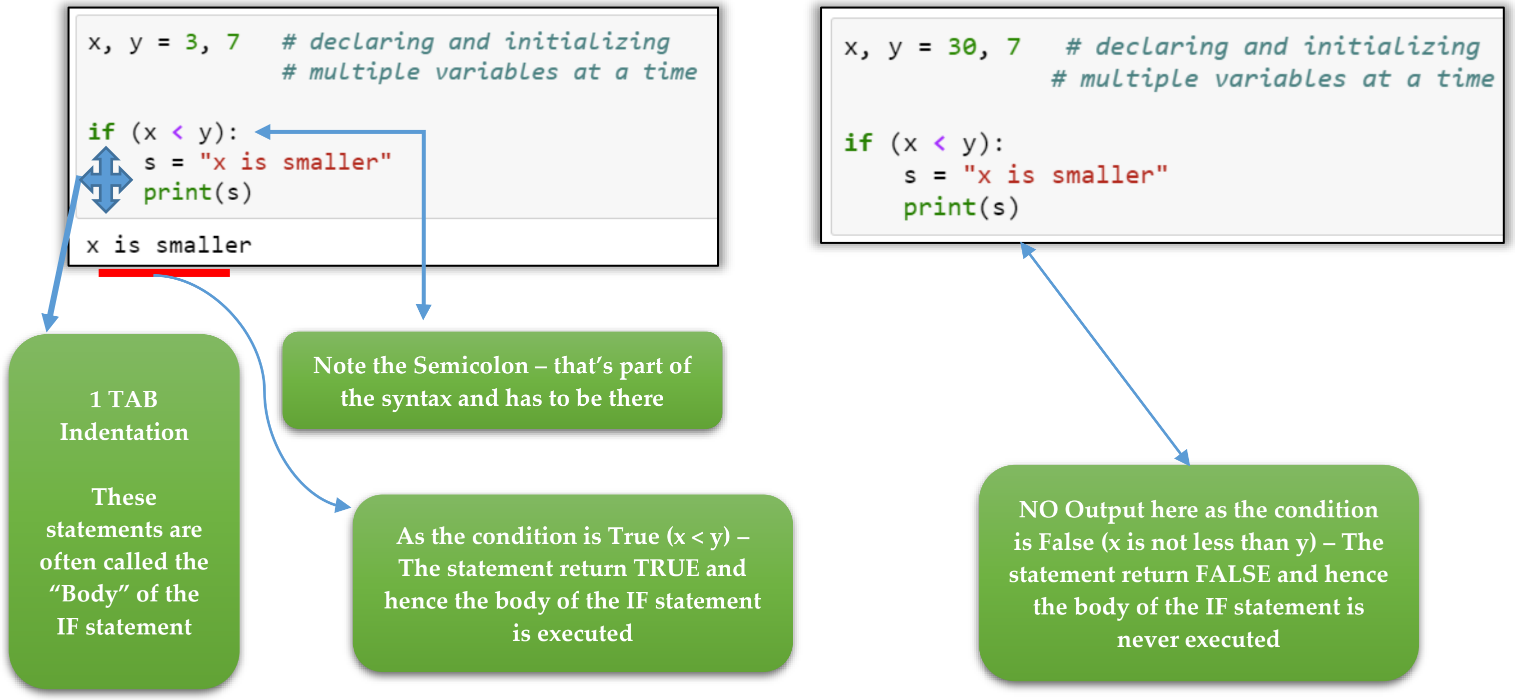
Construct 3 – Conditional Statements (IF, IF…ELSE, ELIF)

Conditional Statements in Python perform different computations or actions depending on whether a specific Boolean constraint evaluates to true or false. Conditional statements are handled by *IF* statements in Python.

Indentation: Indentation is critical to Python. In C\C++ we can use braces - {}, to indicate where a code block starts and ends, However, in case of Python we have to use Indentation as shown in the following code block.

<u>Only 'IF', without 'Else'</u> *<u>[Evaluation of 1 Condition]</u>*

For the program on the left of next page, as you can see, the print statement and the assignment of a string to variable *s* have been right indented (1 TAB), this indicates that all of the code are under the IF statement and will execute if the IF statement returns *True* after the evaluation of the condition. In case the return is a *False*, nothing will be displayed. Execute the program twice by setting the value of x to 3 and then 30.



```
x, y = 3, 7   # declaring and initializing
              # multiple variables at a time

if (x < y):
    s = "x is smaller"
    print(s)
```

```
x is smaller
```

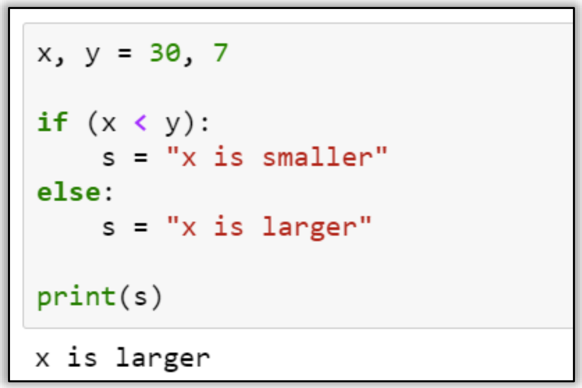
```
x, y = 30, 7   # declaring and initializing
               # multiple variables at a time

if (x < y):
    s = "x is smaller"
    print(s)
```

1 TAB Indentation

These statements are often called the "Body" of the IF statement
Note the Semicolon – that's part of the syntax and has to be there
As the condition is True (x < y) – The statement return TRUE and hence the body of the IF statement is executed
NO Output here as the condition is False (x is not less than y) – The statement return FALSE and hence the body of the IF statement is never executed

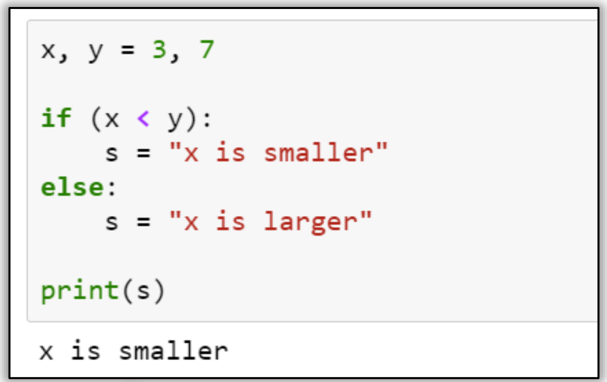
<u>'IF' and 'Else'</u> *<u>[Evaluation of 2 Conditions]</u>*

Now we will add the ELSE condition with the IF. The following program shows how such kind of bock are arranged. The rule of thumb is once the IF statement return FALSE, the ELSE part will be executed, but in case the IF statement returns TRUE, the ELSE segment is ignored.



```
x, y = 30, 7

if (x < y):
    s = "x is smaller"
else:
    s = "x is larger"

print(s)
```

```
x is larger
```



```
x, y = 3, 7

if (x < y):
    s = "x is smaller"
else:
    s = "x is larger"

print(s)
```

```
x is smaller
```

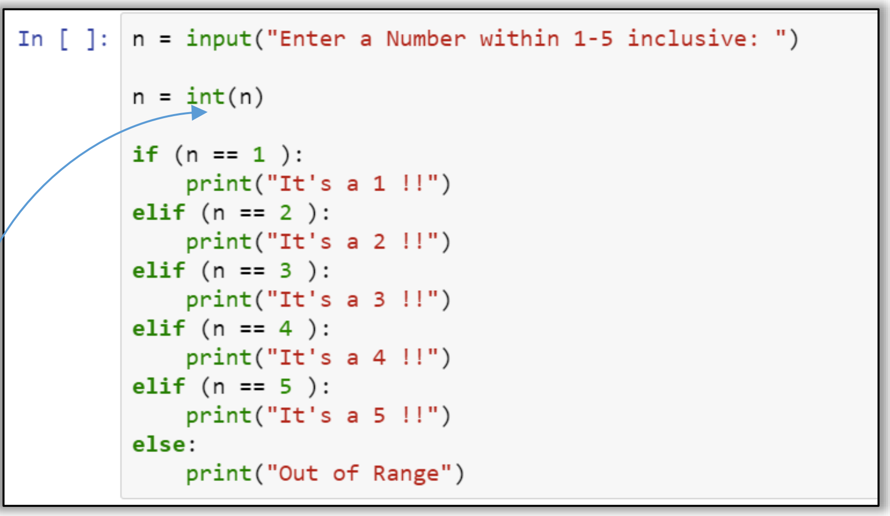
IF statement returned FALSE as x is NOT smaller than y, therefore the ELSE part is executed

IF statement returned TRUE as x is smaller than y, therefore the ELSE part is NOT executed

<u>**'IF' … 'Elif … Else'**</u> <u>***[Evaluation of more than 2 Conditions]***</u>

'Elif' is a Python keyword which is short for 'Else If' and is used to handle multi-conditional scenarios. In the program below the execution is started from the top and one by one all the Elifs are evaluated. If any of the Elifs returns TRUE or the first IF, the body code for that Elif\IF is executed and the program terminates. However, if none of the Elifs and IF condition returns TRUE, the ELSE part is executed.

The code below could have also been written using only IF statements instead of Elif, however, in that case all the IF blocks would have been evaluated even if the outcome is already found in the IFs that have already been executed, basically wasting CPU cycles.



```
In [ ]: n = input("Enter a Number within 1-5 inclusive: ")

        n = int(n)

        if (n == 1 ):
            print("It's a 1 !!")
        elif (n == 2 ):
            print("It's a 2 !!")
        elif (n == 3 ):
            print("It's a 3 !!")
        elif (n == 4 ):
            print("It's a 4 !!")
        elif (n == 5 ):
            print("It's a 5 !!")
        else:
            print("Out of Range")
```

Casting: Casting is done when you convert a variable value from one type to another. This is, in Python, done with functions such as int() or float() or str(). A very common pattern is that you convert a number, currently as a string into a proper number – this is what we have done here. The value that is getting accepted from the user is actually a String and we are converting it to an Integer value.

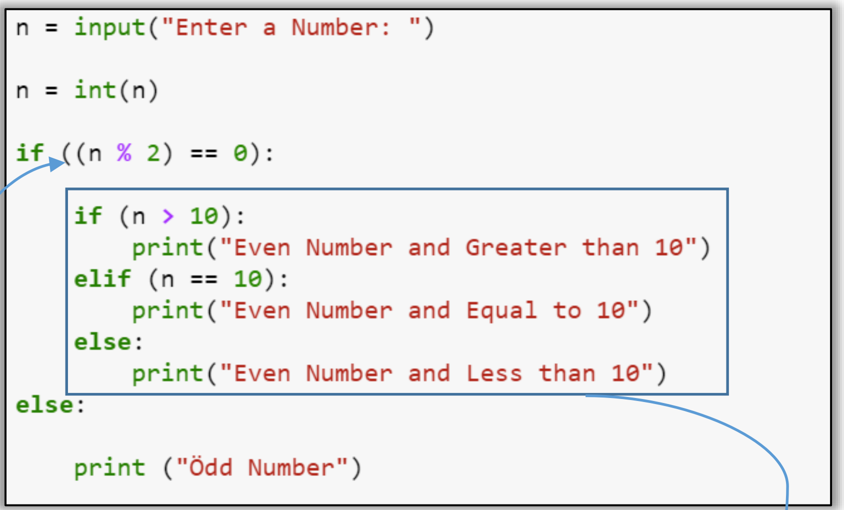
<u>**Also Note:**</u>

The '=' is a simple assignment operator. It assigns values from right side operands to the left side operand. While on the other hand '==' (as in *if n ==1*) checks if the values of two operands are equal or not. If yes, the condition becomes True

The ELSE part is actually *Optional* and you can discard if you are not required to handle values outside 1 – 5; that is if the values are out of range, no message will be shown and the program will simply terminate.

<u>**Nested 'IF' Statements**</u>

IF statements can be nested (one branch of conditional statements are inside another conditional statements) as many levels deeper as required. The following code block shows an example.



```
n = input("Enter a Number: ")

n = int(n)

if ((n % 2) == 0):

    if (n > 10):
        print("Even Number and Greater than 10")
    elif (n == 10):
        print("Even Number and Equal to 10")
    else:
        print("Even Number and Less than 10")
else:

    print ("Ödd Number")
```

The Modulus operator in action (%). The statement checks whether *n* is Even

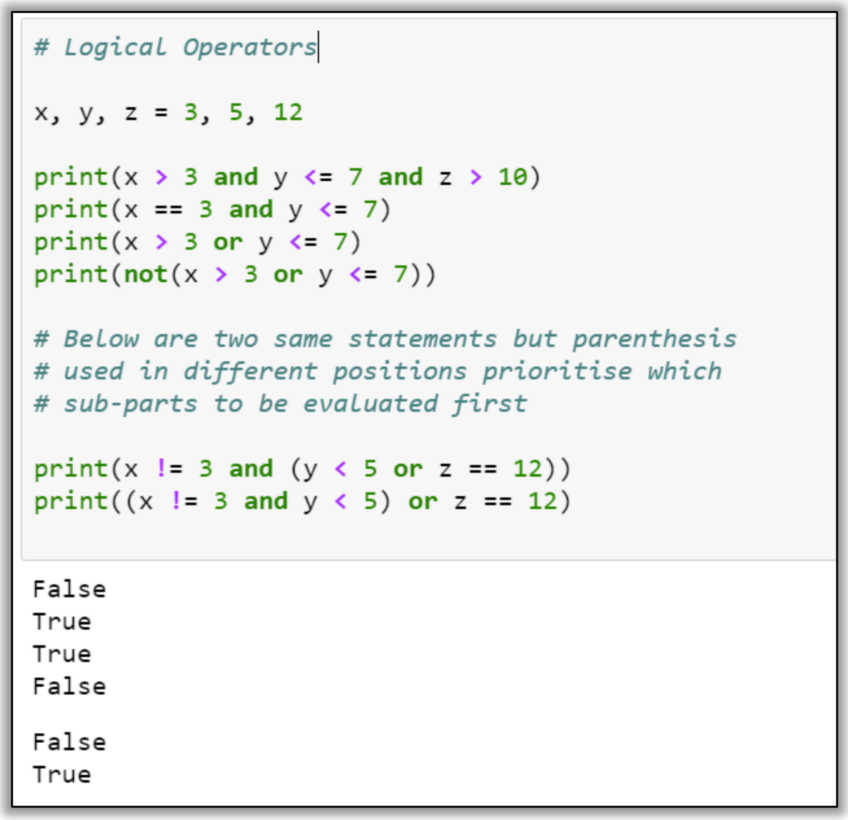
This block of code has been nested inside the first IF statement where we are checking whether the inputted values is Even

Some other necessary operators: **Not Equal (!=), Shorthand (+=, -=, *=, /= ; eg. x = x * 3 is same as x *= 3)**

Logical Operators: These are used to combine conditional statements (***and, or, not***). The final output returned by the full conditional statement in questions is the result of these operators being applied on the results of the individual conditions that it combines.

Test variables: **x** = 3, **y** = 5, **z** = 12

operator	*description*	*example*	*final outcome*
and	Returns TRUE if all sub-statements are True	x > 3 and y <= 7 and z > 10	FALSE
		x = 3 and y <= 7	TRUE
	false		*true*
or	Returns TRUE if one of the sub-statement is True	x > 3 or y < 5 or z = 12	TRUE
not	Reverses the result	not(x > 3 or y <= 7)	FALSE
combining **and** with **or –** you may use parenthesis to priorities sub-parts to be evaluated first		x <> 3 and (y < 5 or z = 12)	FALSE
		(x <> 3 and y < 5) or z = 12	TRUE



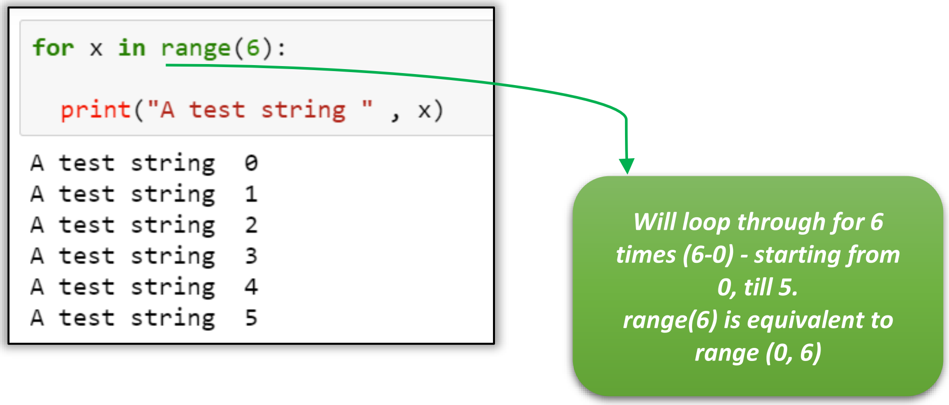
```python
# Logical Operators

x, y, z = 3, 5, 12

print(x > 3 and y <= 7 and z > 10)
print(x == 3 and y <= 7)
print(x > 3 or y <= 7)
print(not(x > 3 or y <= 7))

# Below are two same statements but parenthesis
# used in different positions prioritise which
# sub-parts to be evaluated first

print(x != 3 and (y < 5 or z == 12))
print((x != 3 and y < 5) or z == 12)
```

```
False
True
True
False

False
True
```

Construct 4 – Flow Control (For loop)

Sometimes you would be wanting some part of the code to be executed more than once. We can either repeat the code in our program or use **loops** instead. It is certain that if for instance we need to execute some part of code for a hundred times - it will not be practical to repeat the code exactly hundred times. For such scenarios, loops are necessary and it is also a part of Python's Control Structure repository. The loop we will be investigating is the **'for loop'**.

To loop through a set of code a specific number of times, we can use the ***range()*** function; more about functions in general will be discussed in the later sections. The range() function returns a sequence of numbers, starting from 0 by default, and increments by 1 (by default), and ends at a specified number.

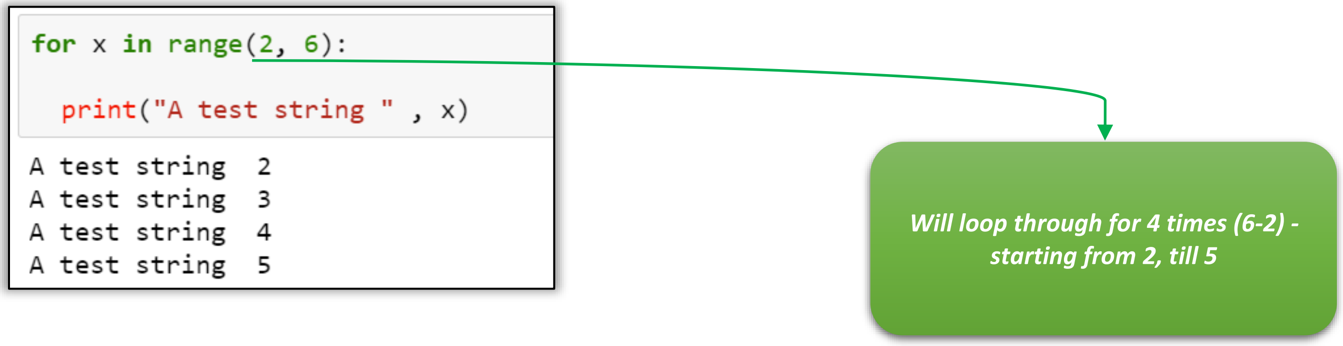


```
for x in range(6):

  print("A test string " , x)
```

```
A test string  0
A test string  1
A test string  2
A test string  3
A test string  4
A test string  5
```

Will loop through for 6 times (6-0) - starting from 0, till 5.
range(6) is equivalent to range (0, 6)

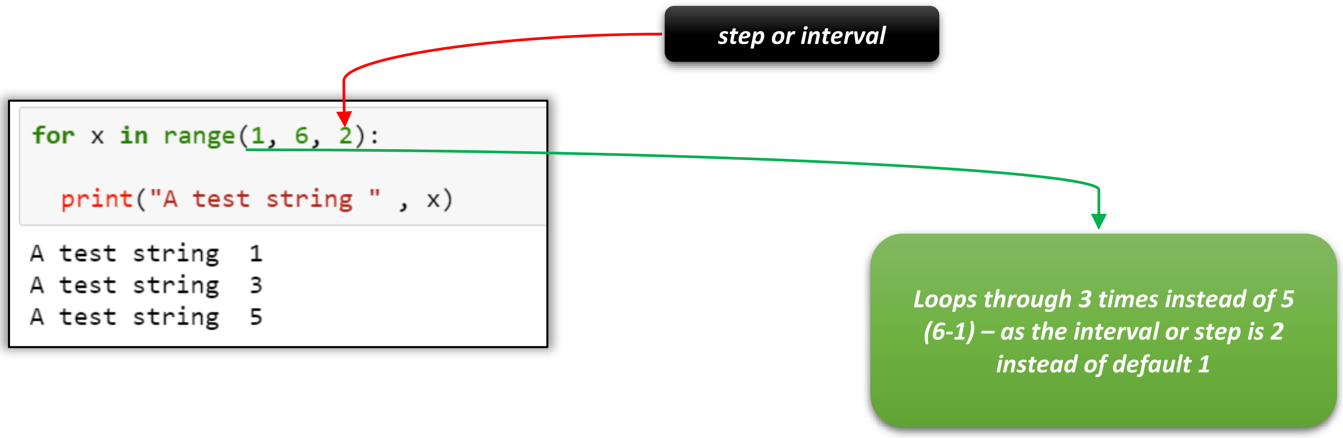
The below two programs shows how differently the range function can be customized to get the desired behavior from the loop body.



```
for x in range(2, 6):

  print("A test string " , x)
```

```
A test string  2
A test string  3
A test string  4
A test string  5
```

Will loop through for 4 times (6-2) - starting from 2, till 5



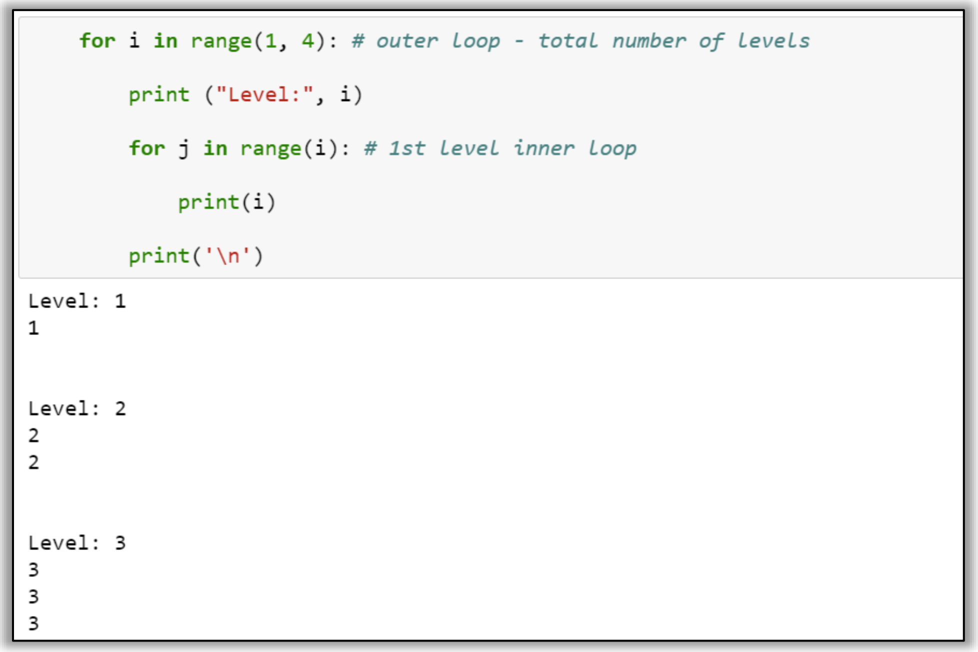
```
for x in range(1, 6, 2):

  print("A test string " , x)
```

```
A test string  1
A test string  3
A test string  5
```

Loops through 3 times instead of 5 (6-1) – as the interval or step is 2 instead of default 1

Nested Looping

Loops can also be nested as demonstrated by the following program. In such cases, once the control is passed from outer loop to the inner one, the inner loop the starts its execution, and it needs to cycle through all the looping before the control goes back to the outer one again.

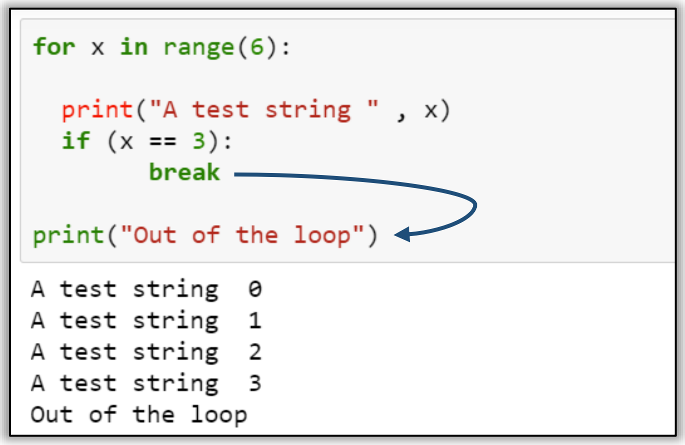


```
for i in range(1, 4): # outer loop - total number of levels

    print ("Level:", i)

    for j in range(i): # 1st level inner loop

        print(i)

    print('\n')
```

```
Level: 1
1


Level: 2
2
2


Level: 3
3
3
3
```

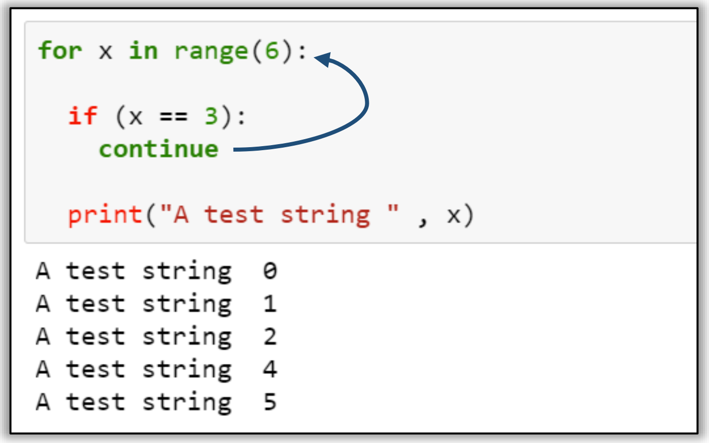
The *break* and *continue* statements

With the **break** statement we can stop the loop before it has looped through all the items; and with the **continue** statement we can stop the current iteration of the loop, sending the control back to the start of the loop. The following two programs demonstrate the concepts.



```
for x in range(6):

  print("A test string " , x)
  if (x == 3):
        break

print("Out of the loop")
```

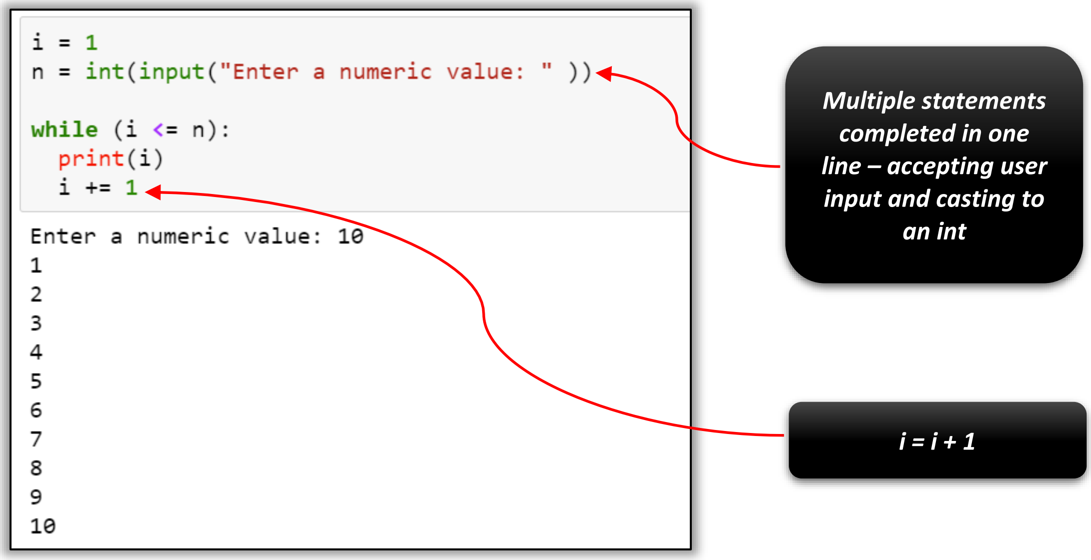
```
A test string  0
A test string  1
A test string  2
A test string  3
Out of the loop
```



```
for x in range(6):

  if (x == 3):
    continue

  print("A test string " , x)
```

```
A test string  0
A test string  1
A test string  2
A test string  4
A test string  5
```

There are other types of Loops as well in python such as 'While Loop'. In such loops the condition is evaluated first and the loop runs as long as the conditions is True.



```
i = 1
n = int(input("Enter a numeric value: " ))

while (i <= n):
  print(i)
  i += 1
```

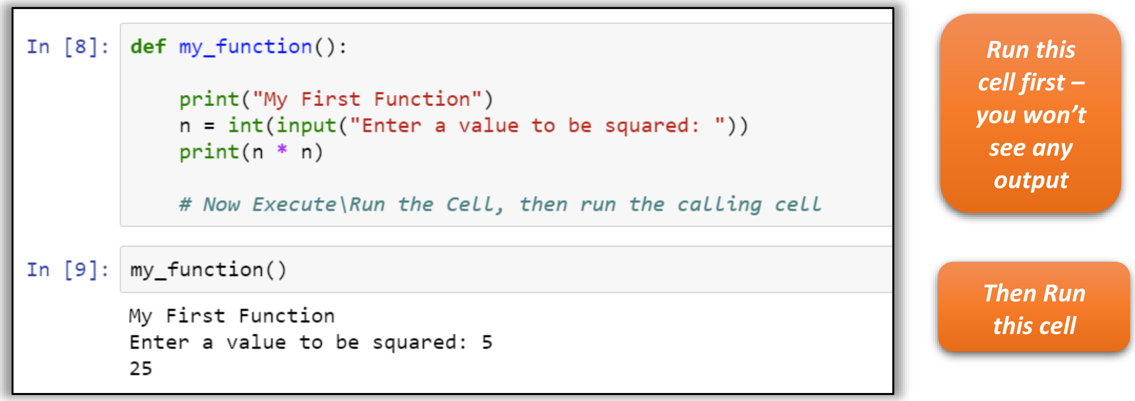
```
Enter a numeric value: 10
1
2
3
4
5
6
7
8
9
10
```

Multiple statements completed in one line – accepting user input and casting to an int
i = i + 1

Construct 5 – Functions

A function is a block of code which only runs when it is called. One can pass data variables, known as parameters, into a function; and a function can also return data as a result. Functions help us in modular programming which not only provides easy manageability but also the ability to reuse parts of the code in other software or programming projects. We use the ***def*** keyword to define a function.

The following image shows a function named 'my_function()' has been placed in cell 8, and the function is called from cell 9.

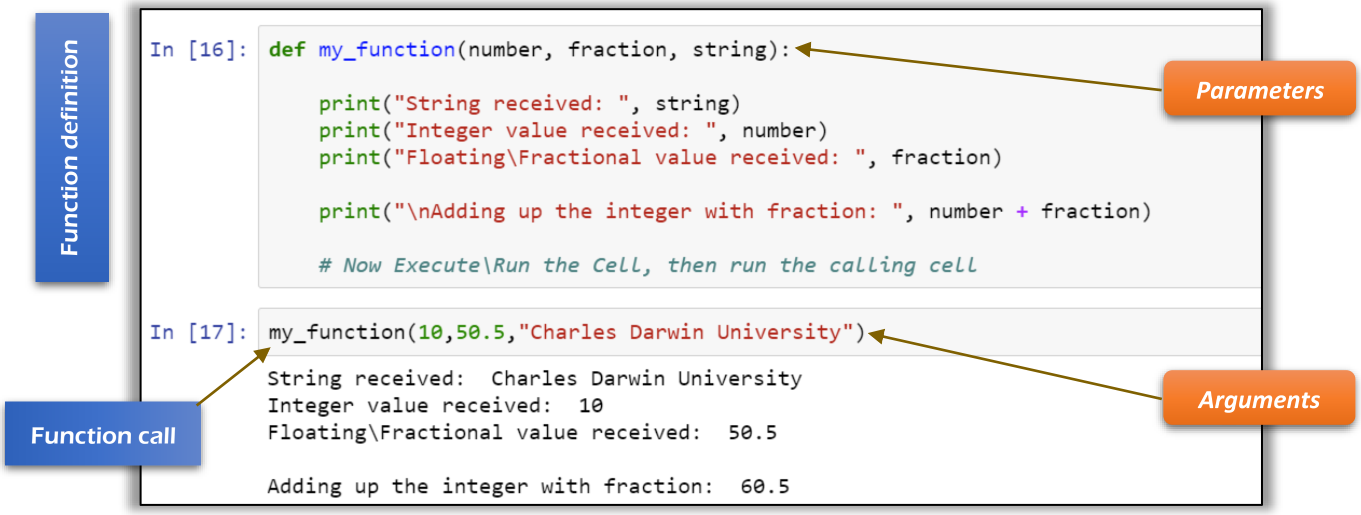


```
In [8]: def my_function():

            print("My First Function")
            n = int(input("Enter a value to be squared: "))
            print(n * n)

            # Now Execute\Run the Cell, then run the calling cell

In [9]: my_function()

        My First Function
        Enter a value to be squared: 5
        25
```

Note that both the function call and the body (definition) of the function could have been placed in the same cell, however, we are showing how placing the function in a separate cell can keep the program tidy. It is important to ensure that before you call the function from cell 9, you Run the cell containing the function body (cell 8 in this case) once you have completed writing the function or have made any changes to it.

Function with Parameters

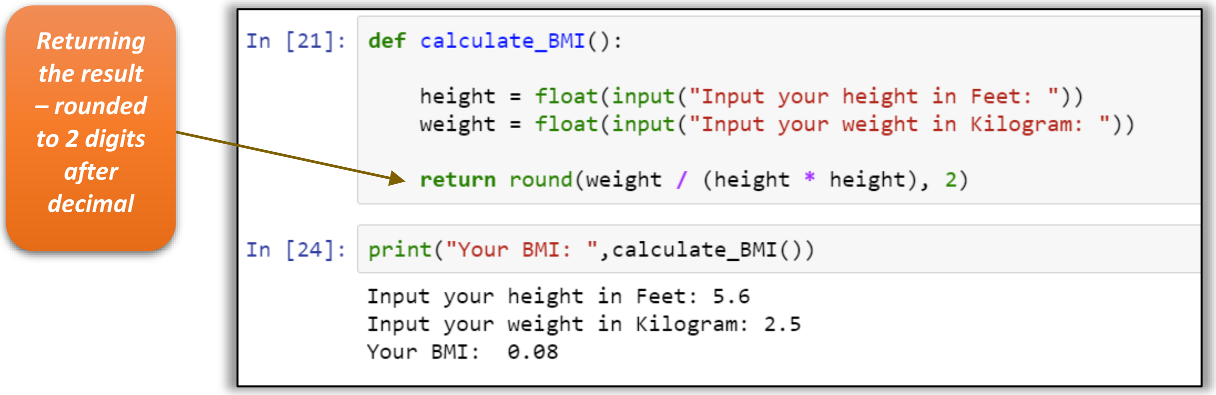
Function can take values – often known as parameters or arguments, and can process those as part of its processing logic. The order of the arguments much the type of parameters defined in the function definition.



```
In [16]: def my_function(number, fraction, string):

             print("String received: ", string)
             print("Integer value received: ", number)
             print("Floating\Fractional value received: ", fraction)

             print("\nAdding up the integer with fraction: ", number + fraction)

             # Now Execute\Run the Cell, then run the calling cell

In [17]: my_function(10,50.5,"Charles Darwin University")

         String received:  Charles Darwin University
         Integer value received:  10
         Floating\Fractional value received:  50.5

         Adding up the integer with fraction:  60.5
```

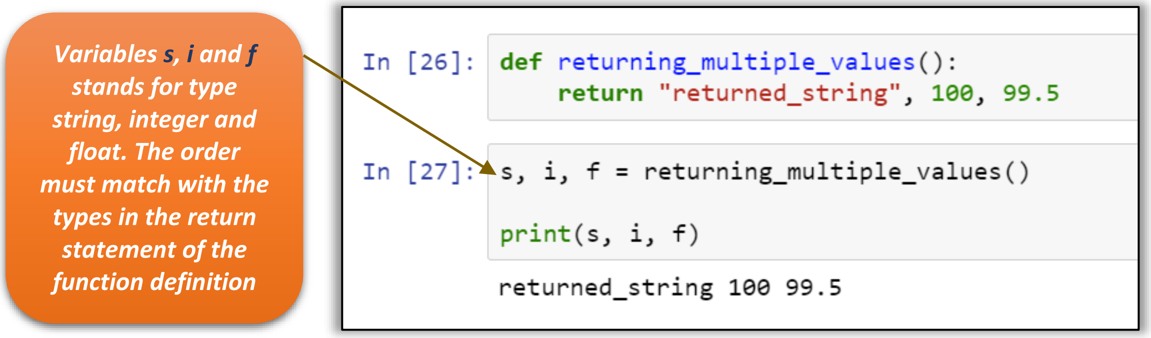
Function returning values

Function can also return values, using the ***return*** keyword, the following program calculates Body-Mass Index using height and weight, and returns the result to the calling segment.



```
In [21]: def calculate_BMI():

             height = float(input("Input your height in Feet: "))
             weight = float(input("Input your weight in Kilogram: "))

             return round(weight / (height * height), 2)

In [24]: print("Your BMI: ",calculate_BMI())

         Input your height in Feet: 5.6
         Input your weight in Kilogram: 2.5
         Your BMI:  0.08
```

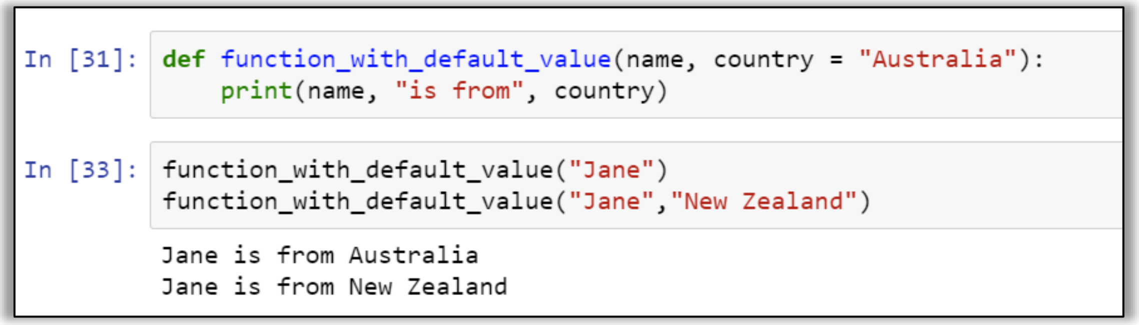
In python, a function can **return multiple values** as demonstrated in the following short program:


Variables s, i and f stands for type string, integer and float. The order must match with the types in the return statement of the function definition

```
In [26]: def returning_multiple_values():
             return "returned_string", 100, 99.5

In [27]: s, i, f = returning_multiple_values()

         print(s, i, f)

         returned_string 100 99.5
```

Default Parameter value

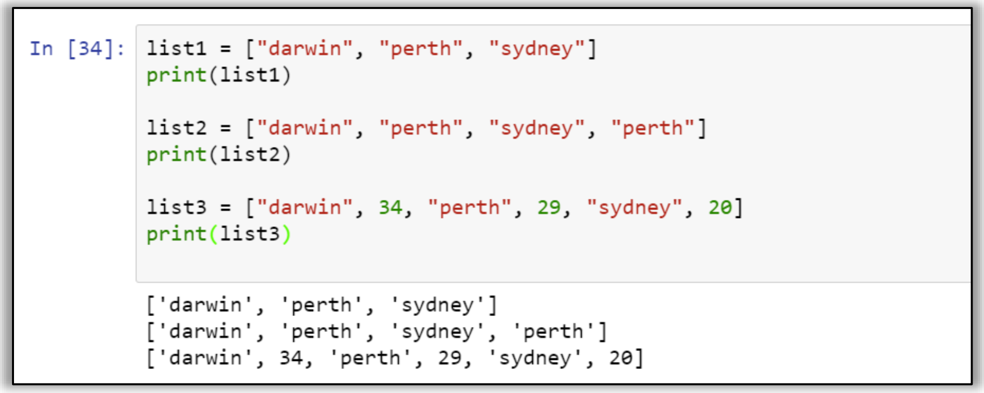
The following example shows how to use a default parameter value. If the function is called without argument, it uses the default value. In the first example we have not passed any argument for 'country' – therefore the default value (Australia) is displayed. However in the second instance a values has been passed (New Zealand), thus that one is used instead the default value. It is useful when you are not sure whether user will enter any argument while calling the function.



```
In [31]: def function_with_default_value(name, country = "Australia"):
             print(name, "is from", country)

In [33]: function_with_default_value("Jane")
         function_with_default_value("Jane","New Zealand")

         Jane is from Australia
         Jane is from New Zealand
```

Construct 6 – Lists in Python

Lists are quite useful and used to store multiple items in a single variable. Lists are one of 4 built-in data types in Python generally used to store collections of data, the other three are **Tuple**, **Set**, and **Dictionary**, all with different qualities and usage.



```
In [34]: list1 = ["darwin", "perth", "sydney"]
         print(list1)

         list2 = ["darwin", "perth", "sydney", "perth"]
         print(list2)

         list3 = ["darwin", 34, "perth", 29, "sydney", 20]
         print(list3)

         ['darwin', 'perth', 'sydney']
         ['darwin', 'perth', 'sydney', 'perth']
         ['darwin', 34, 'perth', 29, 'sydney', 20]
```

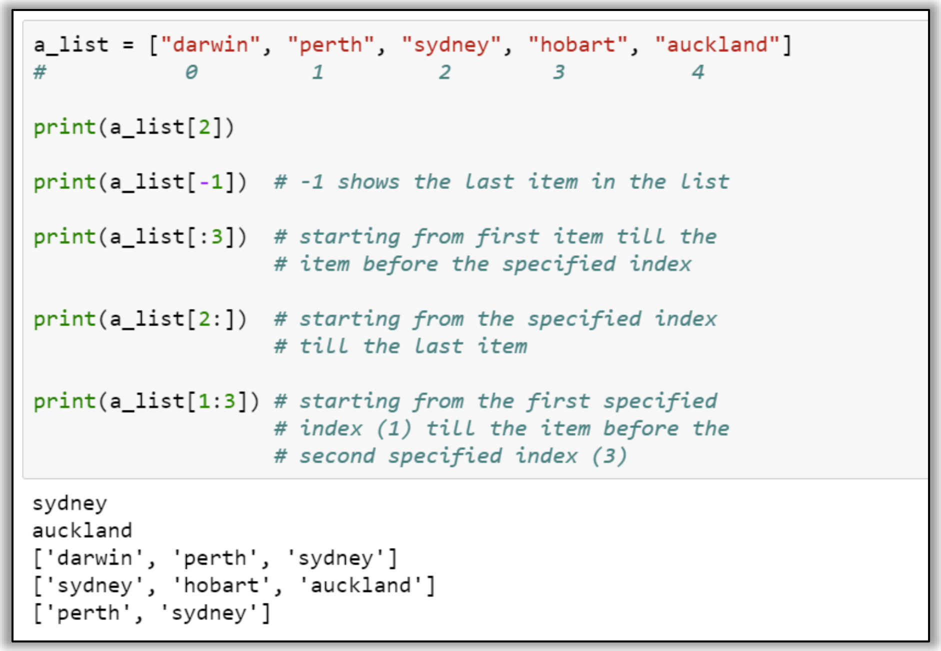
list1 => A normal list of items enclosed in []

list2 => List having duplicate values

list3 => Multiple data types can be stored in a single List

Accessing List Items by Index

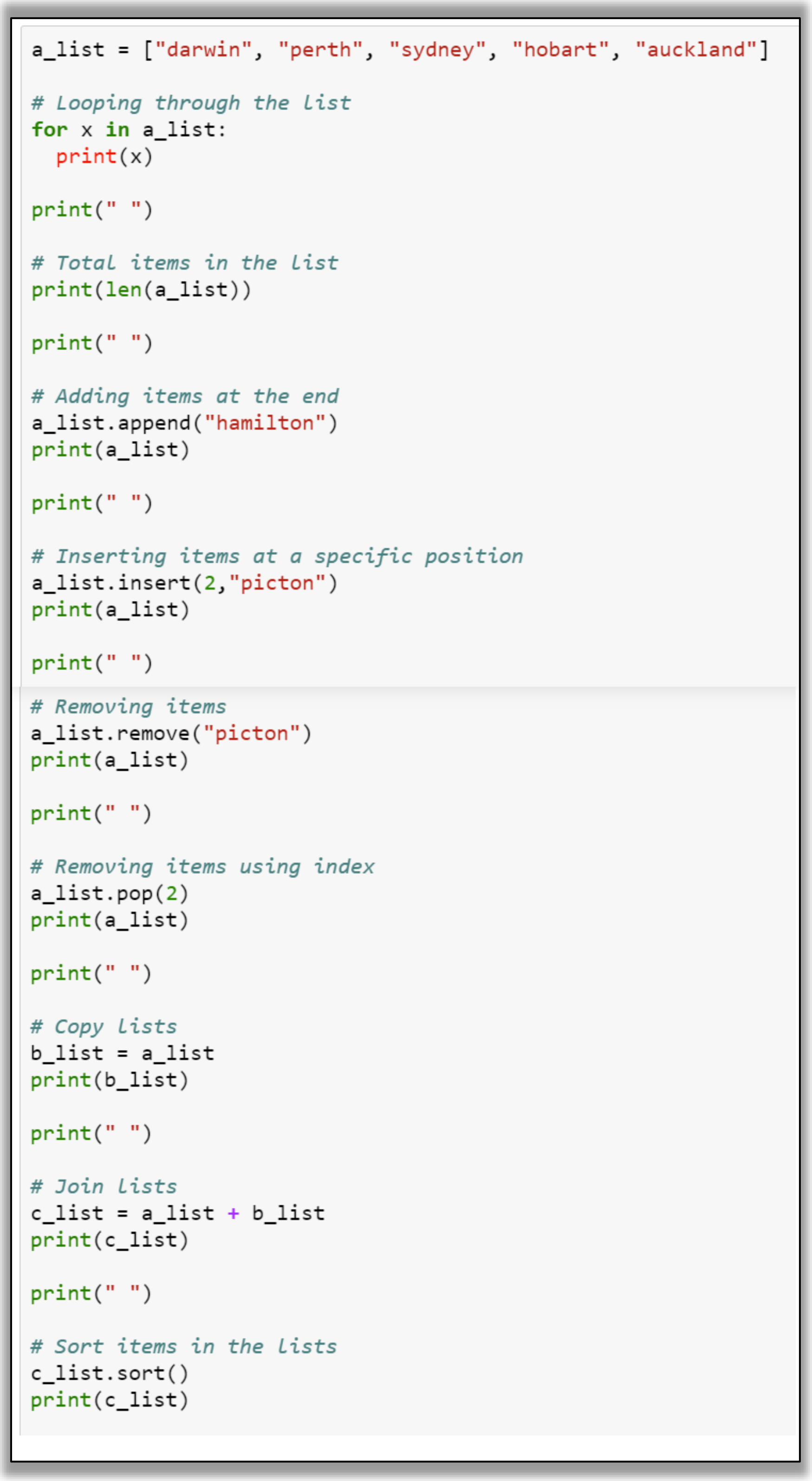
The following block of code shows several way to access list item using index of the items. List Index always starts from 0.



```
a_list = ["darwin", "perth", "sydney", "hobart", "auckland"]
#            0          1         2         3          4

print(a_list[2])

print(a_list[-1])  # -1 shows the last item in the list

print(a_list[:3])  # starting from first item till the
                   # item before the specified index

print(a_list[2:])  # starting from the specified index
                   # till the last item

print(a_list[1:3]) # starting from the first specified
                   # index (1) till the item before the
                   # second specified index (3)

sydney
auckland
['darwin', 'perth', 'sydney']
['sydney', 'hobart', 'auckland']
['perth', 'sydney']
```

Common List operations

Lists can be manipulated in various ways, the below code segment shows an array of such operations that can be performed on a list. Try the codes as shown and inspect the output.

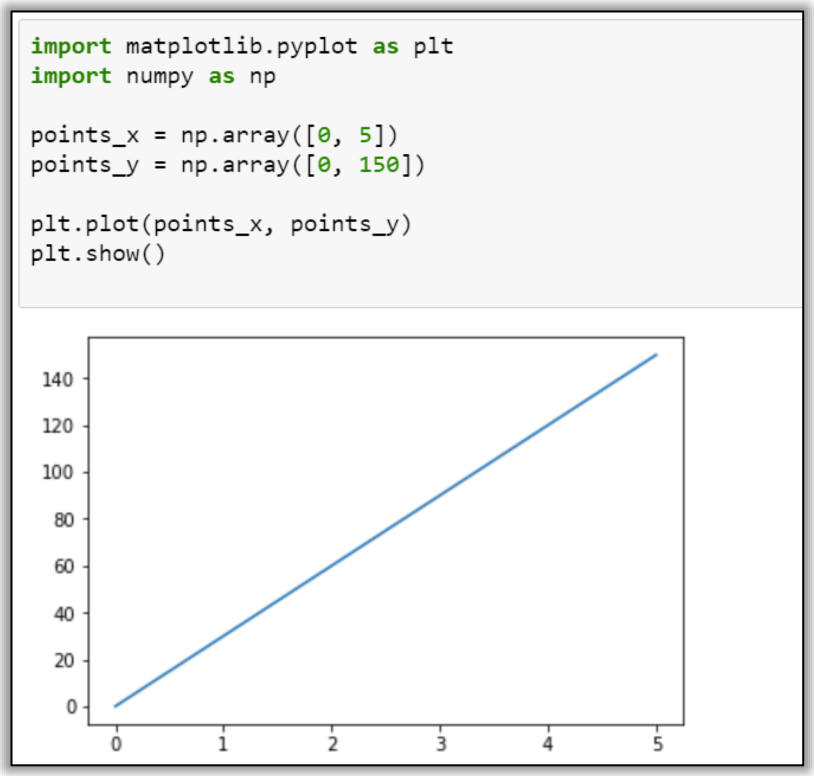


```
a_list = ["darwin", "perth", "sydney", "hobart", "auckland"]

# Looping through the List
for x in a_list:
  print(x)

print(" ")

# Total items in the List
print(len(a_list))

print(" ")

# Adding items at the end
a_list.append("hamilton")
print(a_list)

print(" ")

# Inserting items at a specific position
a_list.insert(2,"picton")
print(a_list)

print(" ")
# Removing items
a_list.remove("picton")
print(a_list)

print(" ")

# Removing items using index
a_list.pop(2)
print(a_list)

print(" ")

# Copy Lists
b_list = a_list
print(b_list)

print(" ")

# Join Lists
c_list = a_list + b_list
print(c_list)

print(" ")

# Sort items in the Lists
c_list.sort()
print(c_list)
```

Construct 7 – Using Libraries through 'import'

Python code in one module gains access to the code in another module by the process of 'importing' it – using the ***import*** statement. Python has a large collection of libraries that lets us use a vast number of pre-built functions in our program.

In this segment we will use one such library called *matplotlib*. Matplotlib is a low level graph plotting library in python that serves as a visualization utility. We will also see how to use *numpy*. NumPy is the fundamental package for scientific computing in Python. This library facilitates a multidimensional array object, various derived objects (such as arrays and matrices), and an assortment of routines for fast operations on arrays.

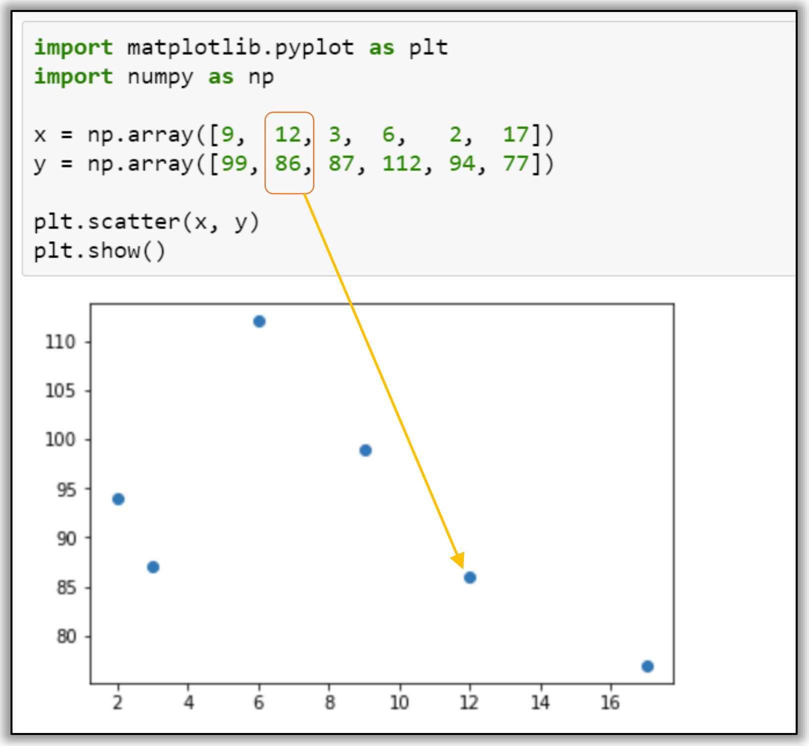


```
import matplotlib.pyplot as plt
import numpy as np

points_x = np.array([0, 5])
points_y = np.array([0, 150])

plt.plot(points_x, points_y)
plt.show()
```

Most of the matplotlib utilities can be found under the *pyplot* submodule, and are usually imported under the *plt* 'alias'. Now the Pyplot package can be referred to as plt. However using alias is optional. We have also used alias for numpy and created two arrays that hold points in the x-axis and y-axis. Eventually the **plot()** function has been called to create the visualization, and **show()** function has been used to display the final plot.

Drawing Scatterplots

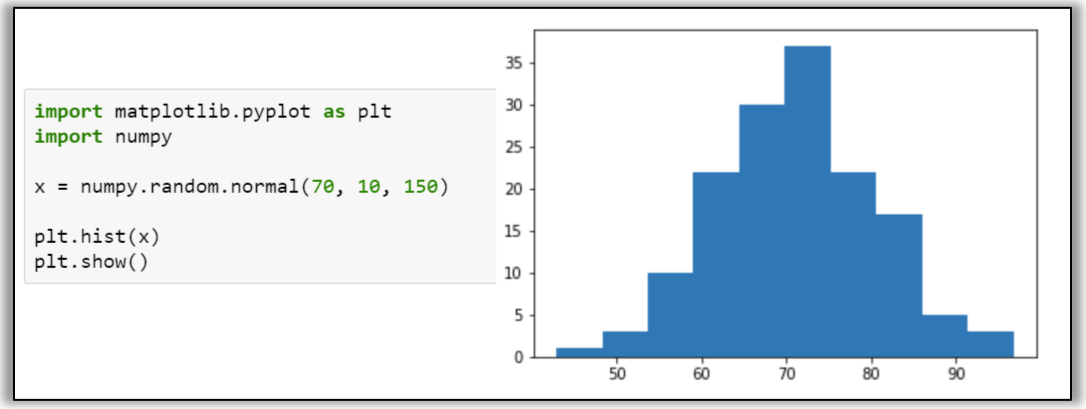
With Pyplot, one can use the **scatter()** function to draw a scatter plot. The **scatter()** function plots one dot for each observation. It requires two arrays of the same length, one for the values of the x-axis, and one for values on the y-axis.



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

x = np.array([9,  12, 3,  6,   2,  17])
y = np.array([99, 86, 87, 112, 94, 77])

plt.scatter(x, y)
plt.show()
```

Drawing Histograms of a Normal Distribution

Using numpy, without alias this time, we randomly generated an array with 150 values, where the values will concentrate around 70, and the Standard Deviation will be 10. The **hist()** function takes this array as an argument and produces the desired Histogram.



```
import matplotlib.pyplot as plt
import numpy

x = numpy.random.normal(70, 10, 150)

plt.hist(x)
plt.show()
```

Additional Resources

The video lecture materials on Python has detailed explanation on a wide range of topic related to Python including the language constructs discussed in this tutorial. You can download all the videos from: https://drive.google.com/file/d/1nucoYZtU3j5Nti2sqp_j2XhMgh5SiQuD/view?usp=sharing

Password to unzip (1.75 GB of zip file): **prt565**

Below are a brief description of the video content –

filename	*description*	*filename*	*description*
Video_1	general discussion on python including basic history and the program compilation process	**Video_22**	programming example using functions
Video_2	setting up of ATOM editor, can be ignored for this course as we are using Jupyter	**Video_23**	recursive functions
Video_3	operators, variables, strings and lists	**Video_24**	programming examples
Video_4	if … then … else\elif - discussion	**Video_25**	programming examples
Video_5	if … then … else\elif - tutorial	**Video_26**	programming examples
Video_6	data types, variables, set, lists, tuple, random number generator	**Video_27**	programming examples
Video_7	boolean logic, for loop	**Video_28**	graphics operations
Video_8	programming examples	**Video_29**	graphics programming examples
Video_9	programming using while loops	**Video_30**	graphics programming examples
Video_10	logical operators, string, file handling	**Video_31**	graphics programming examples
Video_11	data encryption and string methods	**Video_32**	linear and binary searching
Video_12	reading from text files	**Video_33**	sorting algorithms
Video_13	arrays	**Video_34**	sorting algorithms
Video_14	general file operations including writing data to text files	**Video_35**	object oriented programming
Video_15	list operations	**Video_36**	object oriented programming
Video_16	list operations and tuples	**Video_37**	object oriented programming
Video_17	programming example using lists	**Video_38**	operator overloading
Video_18	functions	**Video_39**	GUI development
Video_19	dictionaries	**Video_40**	GUI development
Video_20	programming example using dictionaries	**Video_41**	GUI development
Video_21	functions	**Video_42**	GUI development - event

========================= X =========================