



## RMB-PYTHON-2D-ONL-101 - Session 01 - Intro to Python Basics

**24 November 2022**

by Philip Booyen <[philip@piguy.co.za](mailto:philip@piguy.co.za) (mailto:[philip@piguy.co.za](mailto:philip@piguy.co.za))>

### Table of contents

#### A. Introduction to Python

1. [Intro Python](#)
2. [Intro Jupyter Notebook](#)
3. [Becoming fluent in Jupyter Notebook](#)
4. [Let's get into the Python Basics](#)
5. [Print Statement](#)
6. [Comments](#)
7. [Variables](#)
8. [Strings](#)
9. [Numbers, String and Boolean Types](#)
10. [Collections of data - Lists and Dictionaries](#)
11. [User-define Functions](#)
12. [Control Flow Statements](#)

#### B. Homework

1. [Review this Jupyter Notebook](#)
2. [Python Pandas Video](#)
3. [Python xlwings Video](#)
4. [While Loop Excercise](#)
5. [Functions Excercise](#)

## 1. Intro Python

### 1.1 What is Python

Python is a general purpose programming language created in the late 1980s, and named after Monty Python, that's used by thousands of people to do things from testing microchips at Intel, to powering Instagram, to building video games with the PyGame library.

### 1.2 History

Python was born in December of 1989, when Guido van Rossum from Netherlands created Python while being on his Christmas break while working for Stichting Mathematisch Centrum. He helped create the ABC programming language in 1986, but wanted to improve on ABC and came up with Python.

### 1.3 What is in a name?

Being in an irreverent mood that week of Xmas '89, Van Rossum named "Python" after the British TV sketch comedy show "Monty Python's Flying Circus".

### 1.4 Most Famous Software Programs Written in Python

- YouTube
- DropBox
- Google
- Quora
- Instagram
- Spotify

[\[Back to top\]](#)

## **2. Intro Jupyter Notebook**

### **2.1 What is Jupyter Notebook**

The Jupyter Notebook is an open-source web application that allows you to create and share documents that contain live code, equations, visualizations and narrative text. Uses include: data cleaning and transformation, numerical simulation, statistical modeling, data visualization, machine learning, and much more.

### **2.2 History and Name**

The Notebook interface was added to IPython in December 2011, renamed to Jupyter notebook in 2015 (IPython 4.0 – Jupyter 1.0).

### **2.3 What makes it popular**

- Ability to re-run portions of a program, rather than the entire thing
- Very convenient for experimenting with big datasets
- Very easy to explore idea and tests code
- Easiness of sharing
- Make it easy to work on graphical data, interactive graphs and even SQL results

### **2.4 As preferred interpreter**

Clearly Jupyter Notebook makes it possible for a live manual during training, and can be used as the preferred Python interpreter and development interface even beyond this context.

[\[Back to top\]](#)

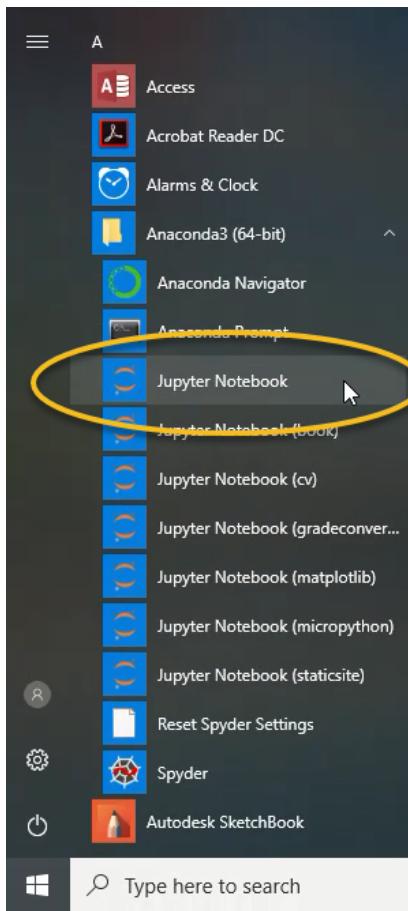
## **3. Becoming fluent in Jupyter Notebook**

### **3.1. How to open and run Jupyter Notebook**

**Prerequisites:** Make sure Anaconda was installed on your Windows 10

To start Jupyter Notebook, do the following on your Laptop in Windows 10:

1. Open the Windows start menu and select [Anaconda3(64 bit)] → [Jupyter Notebook]



3. Wait for Jupyter Notebook to start up, a console with the backend will start up
4. Your default browser should open up with a new tab pointing to localhost:8888/tree

Once up and running, **Jupyter Notebook is now running** in your browser:

A screenshot of a web browser window displaying the Jupyter Notebook interface. The address bar shows 'localhost:8888/tree'. The main page title is 'jupyter'. Below the title, there are tabs for 'Files', 'Running', and 'Clusters'. The 'Files' tab is selected. A file list table is shown with the following data:

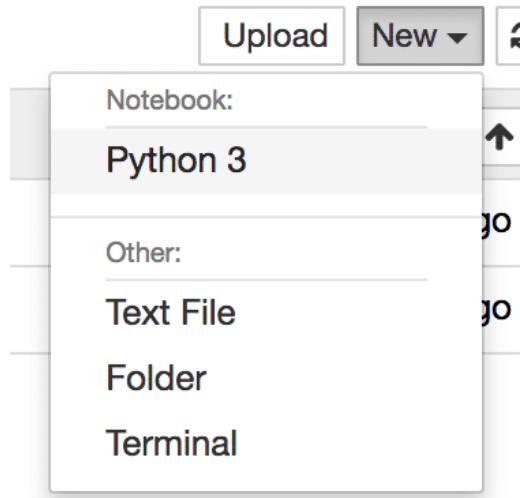
	Name	Last Modified
0	/	
■	Anaconda3	5 days ago
■	AnacondaProjects	5 days ago
■	AppData	3 years ago
■	Contacts	3 years ago
■	Desktop	8 minutes ago

## 3.2. How to use Jupyter Notebook

### 3.2.1 Creating A New Notebook

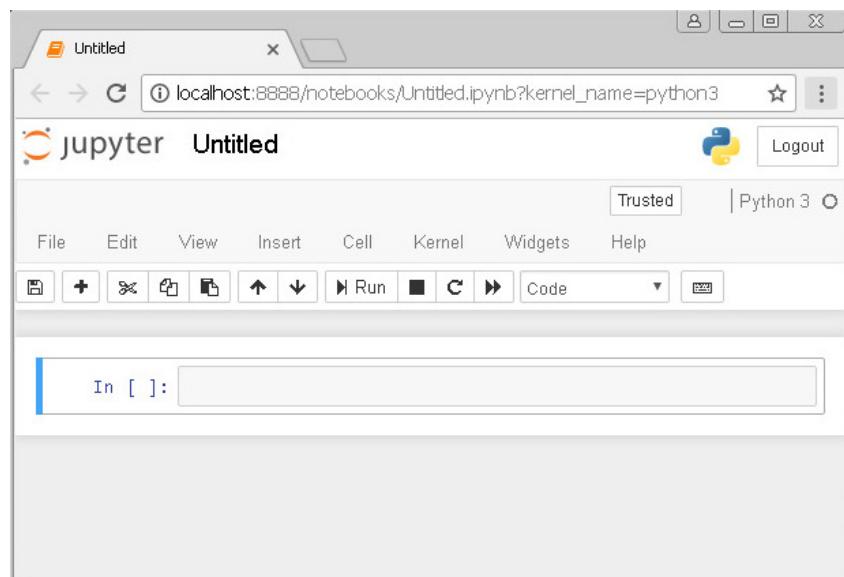
To create a new Notebook:

- Make sure you are in the "Files" tab
- Then use the "New" dropdown menu and you'll see the following options:



Select the "Python 3" option to open a new Jupyter Notebook for Python.

The notebook gets created and you should be able to now see something similar to the following:

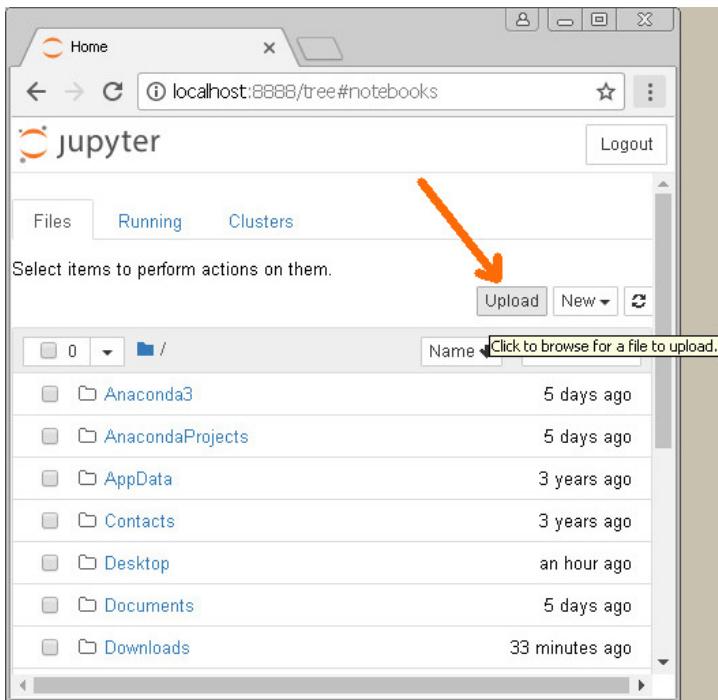


### 3.2.2 Uploading an existing Notebook

In the files section, you could browse to a location where there is a downloaded Python Notebook (.ipynb).

You can also **upload** an existing Notebook as follows:

- Make sure you are in the "**Files**" tab
- Then click the "**Upload**" button
- Select the .ipynb file to upload and **click "Open"**:



### 3.2.3 Working With This "Pi Guy" Python Training Notebook

Ensure you have **downloaded the RMB-PYTHON-ONL-102-01-Intro-Python.ipynb** file which the Pi Guy training instructor supplied together with the PDF version of this manual you are reading now.

Make sure you follow the steps outlined in the previous section on "**Uploading an existing Notebook**" and load the **RMB-PYTHON-ONL-102-01-Intro-Python.ipynb**

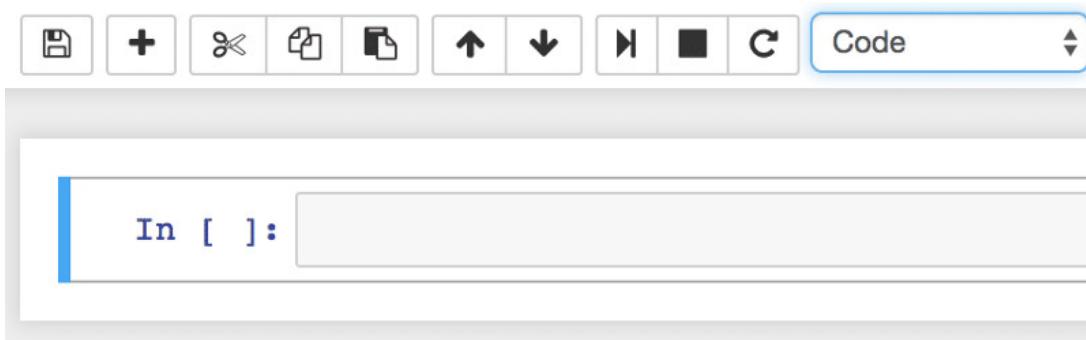
From this point onwards in this Notebook, you can interact live with Python code.

Follow and interact with the rest of this document by using Python Notebook (not the PDF).

It is imperative to get some practical hands-on experience, so ensure you carry out the rest of this preparation in Jupyter Notebook.

### 3.2.4 Working With The Notebook

The notebook itself consists of cells. A first empty cell is already available after having created the new notebook:



This cell is of type "Code" and you can start typing in Python code directly. Executing code in this cell can be done by either clicking on the run cell button or hitting SHIFT + ENTER keys:

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

The resulting output becomes visible right underneath the cell.

If you would use "ALT + ENTER" it would execute and create a new cell for you to write some more code in.

This should make the output look exactly like the figure above.

Try out yourself here under, click inside the code block, and press ALT + ENTER to execute it and create a new cell to code in:

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

Hello World

You can also just run a piece of Python code in a block, as mentioned earlier, by hitting SHIFT + ENTER. This will not create a new cell, just execute the code.

Try and change "Hello World" above to a different string, and then hit SHIFT + ENTER.

### 3.2.5 Edit And Command Mode

If a cell is active two modes distinguished:

- edit mode
- command mode

If you just click in one cell the cell is opened in **command mode** which is indicated by a **blue border** on the left:

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

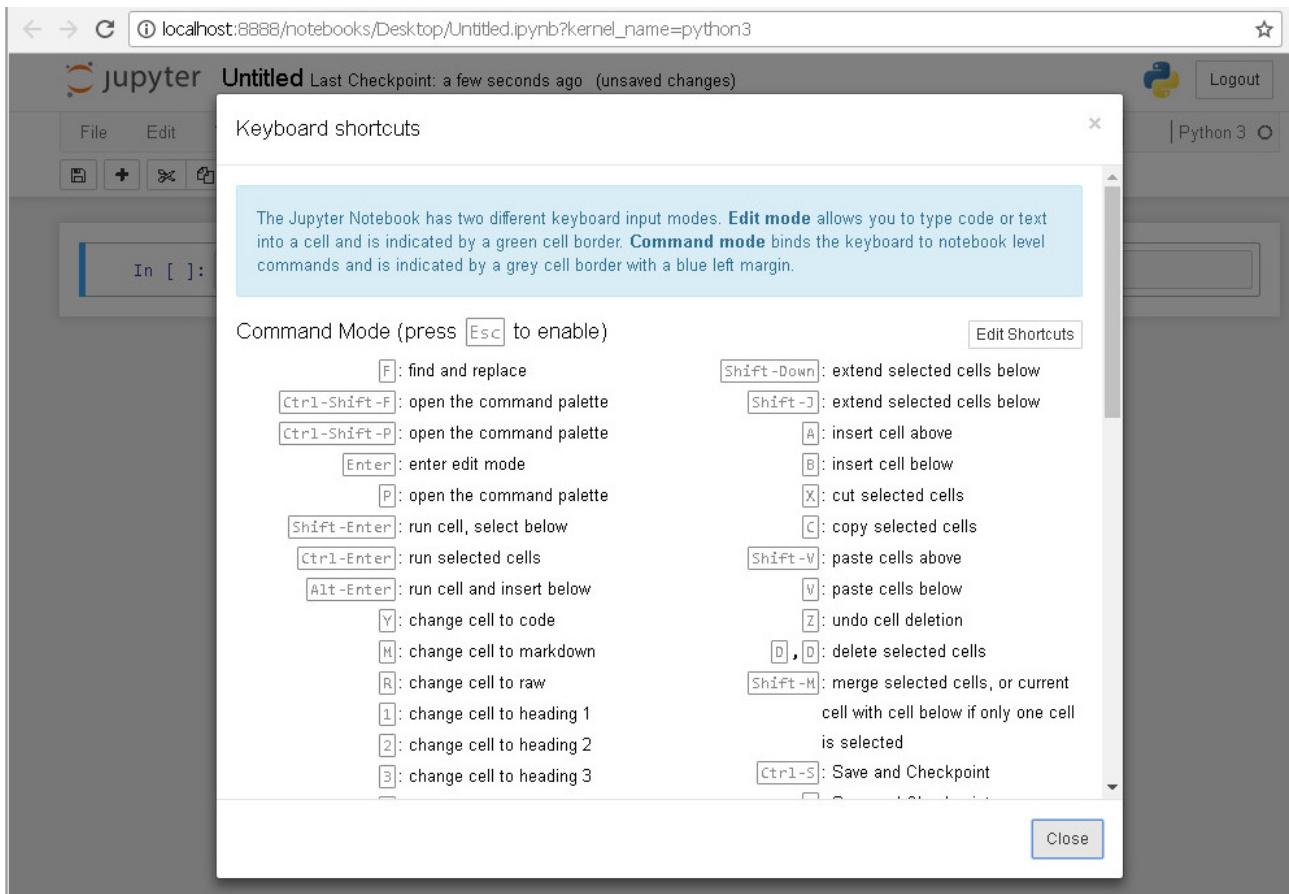
The **edit mode** is entered if you click into the code area of that cell. This mode is indicated by a **green border** on the left side of the cell:

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

If you'd like to leave edit mode and return to command mode again you just need to hit ESC.

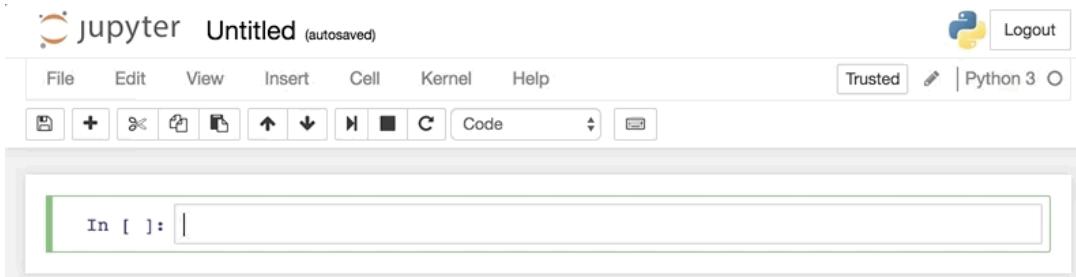
### 3.2.6 Keyboard Shortcuts

To get an overview of functions which are available in command and in edit mode you can open up the overview of key shortcuts by using menu entry *Help → Keyboard Shortcuts* or by just hitting the key "H" in command mode:



### 3.2.7 How to use Jupyter Notebook (Re-cap)

Let's recap how to use Jupyter Notebook



Type your Python command! It can be a multi-line command too – if you hit return/enter, it won't run, it will just start a new line in the same cell!

```
In [ ]: print('Hello, World!')
print('This is just nice!')
```

Hit SHIFT + ENTER to run your Python command!

```
In [1]: print('Hello, World!')
print('This is just nice!')

Hello, World!
This is just nice!
```

Start typing and hit TAB! If it's possible, Jupyter will auto-complete your expression (eg. for Python commands or for variables that you have already

defined). If there is more than one possibility, you can choose from a drop-down menu.

```
In [ ]: pr
  %precision
  print
  %profile
  property
  %prun
  %%prun
```

```
In [2]: print("Python is easy to learn")
```

Python is easy to learn

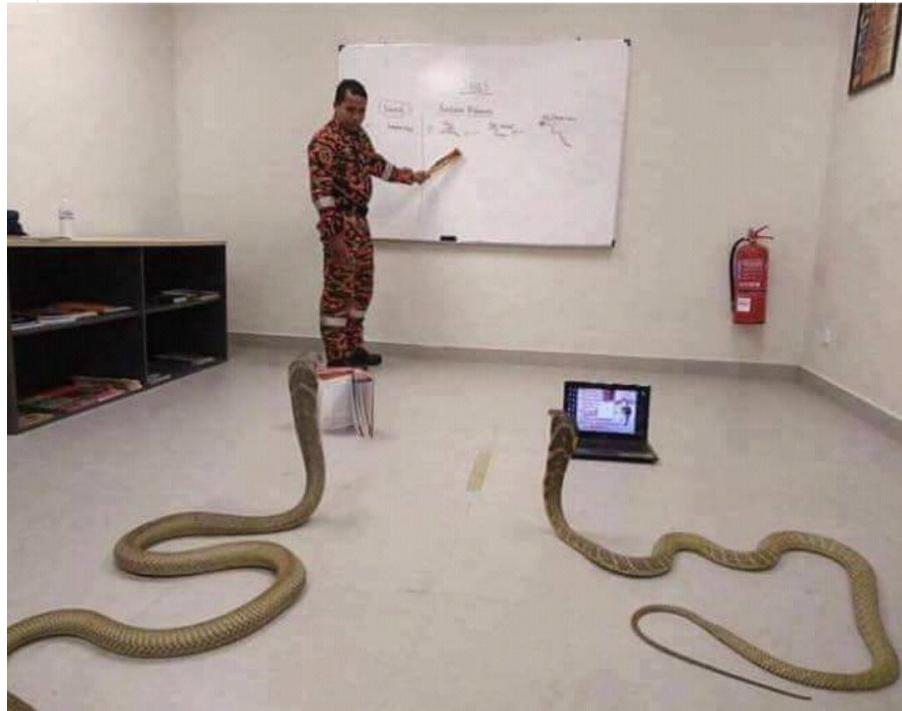
[\[Back to top\]](#)

## 4. Let's get into the Python Basics

Learning Python programming is intense



\*yes, those are Cobras\*



The rest of this section delves into the Basics on Python, but just before we do, let's take a quick look at:

- Where can you find documentation
- How to import extra functionality using "import"
- How to get help using "dir"
- How to get help using "help"

### 4.1 Official Python docs

Official Python Documentation can be found at: <https://docs.python.org/3/>

This includes:

- Tutorial: <https://docs.python.org/3/tutorial/index.html>
- Python Language Reference: <https://docs.python.org/3/reference/index.html>

- Python HOWTOs: <https://docs.python.org/3/howto/index.html>
- FAQs: <https://docs.python.org/3/faq/index.html>

[[Back to top](#)]

## 5. Print Statement

Print displays output to your console, or screen, or Jupyter Notebook "output" below a cell.

The `print` statement or function can be used in the following different ways :

- `print ("Hello World")`
- `print ("Hello", <Variable Containing the String>)`
- `print ("Hello" + <Variable Containing the String>)`

When programmers are learning a new language, we tend to write a one-line program that prints some version of the message "Hello world!"

```
In [3]: print ("Hello World!")
```

Hello World!

We need to show Python we are working with a **text string** by **enclosing the text in quotes**.

You can also think of a string as a string of characters.

In Python, single, double and triple quotes are used to denote a string.

There is **no difference between single or double quotes**, you can choose what to use.

Most use single quotes when declaring a single character.

Double quotes when declaring a line and triple quotes when declaring a paragraph/multiple lines.

```
In [4]: print ('Hey')
```

Hey

```
In [5]: print ("""My name is the Pi Guy  
I love Python.""")
```

My name is the Pi Guy

I love Python.

Strings can be assigned to variable say `string1` and `string2` which can be called when using the `print` statement.

```
In [6]: string1 = 'World'  
print ('Hello', string1)  
  
string2 = '!'  
print ('Hello', string1, string2)
```

Hello World  
Hello World !

String concatenation is the "addition" of two strings. Observe that while concatenating there will be no space between the strings.

```
In [7]: print ('Hello' + string1 + string2)
```

HelloWorld!

### Print In-class Exercise

The `input` statement or function helps you to **get information from a user**

As part of this simplified exercise, make sure you can:

1. Run the following code cell below
2. Once running, provide your name in the text box
3. Press ENTER after you provided your name
4. See the output created by the "print" statement

```
In [8]: # The input statement helps us to get input from the user
answer=input("Please enter your name: ")

# Then the response got stored into the variable "answer", which we can use below:
print("Welcome " + answer + "!")

Please enter your name: Pi Guy
Welcome Pi Guy!
```

[\[Back to top\]](#)

## 6. Comments

As you begin to write more complicated code, you will have to spend more time thinking about how to code solutions to the problems you want to solve. Once you come up with an idea, you will spend a fair amount of time troubleshooting your code, and revising your overall approach.

Comments allow you to write in English, within your program. In Python, any line that starts with a hash (#) symbol is ignored by the Python interpreter.

Note that the comment ends at the end of the physical line.

```
In [9]: # This line is a comment.
print("This line is not a comment, it is code.")
```

This line is not a comment, it is code.

### 6.1 What makes a good comment?

- It is short and to the point, but a complete thought. Most comments should be written in complete sentences.
- It explains your thinking, so that when you return to the code later you will understand how you were approaching the problem.
- It explains your thinking, so that others who work with your code will understand your overall approach to a problem.
- It explains particularly difficult sections of code in detail.

### 6.2 When should you write comments?

- When you have to think about code before writing it.
- When you are likely to forget later exactly how you were approaching a problem.
- When there is more than one way to solve a problem.
- When others are unlikely to anticipate your way of thinking about a problem.

Writing good comments is one of the clear signs of a good programmer.

Start using comments now. You will see them throughout the examples in these notebooks.

It ultimately also helps you or anyone else that has to understand, improve, modify or use your code.

[\[Back to top\]](#)

## 7. Variables

In Python we like to assign values to variables.

Variables just holds values or certain data for us.

Some of the best reasons it because it makes our code better - more flexible, reusable and understandable.

### 7.1 Variable assignment

At the same time one of the trickiest things in coding is exactly this "assignment concept".

A name that is used to denote something or a value is called a variable. In python, variables can be declared and values can be assigned to it as follows:

```
In [10]: x = 3
```

```
In [11]: print(x)
```

3

```
In [12]: my_number = 5
```

```
In [13]: Surname = 'Monty'
```

```
In [14]: print(x+my_number, Surname)
```

```
8 Monty
```

Multiple variables can be assigned with the same value.

```
In [15]: x = y = 1
```

```
In [16]: print (x)
```

```
1
```

You can change the value of a variable at any point.

```
In [17]: message = "Hello Python world!"
print(message)

message = "Python is my favorite language!"
print(message)
```

```
Hello Python world!
Python is my favorite language!
```

## 7.2 Naming rules

- Variables can only contain letters, numbers, and underscores. Variable names can start with a letter or an underscore, **but can not start with a number**.
- **Spaces are not allowed** in variable names, so we use underscores instead of spaces. For example, use student\_name instead of "student name".
- You cannot use [Python keywords](http://docs.python.org/3/reference/lexical_analysis.html#keywords) ([http://docs.python.org/3/reference/lexical\\_analysis.html#keywords](http://docs.python.org/3/reference/lexical_analysis.html#keywords)) as variable names.
- Variable names should be descriptive, without being too long. For example mc\_wheels is better than just "wheels", and number\_of\_wheels\_on\_a\_motorcycle.
- Be careful about using the lowercase letter l and the uppercase letter O in places where they could be confused with the numbers 1 and 0.

## 7.3 NameError

There is one common error when using variables, that you will almost certainly encounter at some point. Take a look at this code, and see if you can figure out why it causes an error.

```
In [18]: message = "Thank you for sharing Python with the world, Guido!"
print(mesage)
```

```
-----  
NameError                                 Traceback (most recent call last)
<ipython-input-18-85929e5c0a3c> in <module>
      1 message = "Thank you for sharing Python with the world, Guido!"
----> 2 print(mesage)
```

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

Let's look through this error message. First, we see it is a NameError. Then we see the file that caused the error, and a green arrow shows us what line in that file caused the error. Then we get some more specific feedback, that "name 'mesage' is not defined".

You may have already spotted the source of the error. We spelled message two different ways. Python does not care whether we use the variable name "message" or "mesage". Python only cares that the spellings of our variable names match every time we use them.

This is pretty important, because it allows us to have a variable "name" with a single name in it, and then another variable "names" with a bunch of names in it.

We can fix NameErrors by making sure all of our variable names are spelled consistently.

```
In [19]: message = "Thank you for sharing Python with the world, Guido!"
print(message)
```

```
Thank you for sharing Python with the world, Guido!
```

As mentioned earlier, [Guido](http://en.wikipedia.org/wiki/Guido_van_Rossum) ([http://en.wikipedia.org/wiki/Guido\\_van\\_Rossum](http://en.wikipedia.org/wiki/Guido_van_Rossum)) [van Rossum](http://www.python.org/~guido/) (<http://www.python.org/~guido/>) created the Python language almost 20 years ago, and he is considered Python's [Benevolent Dictator for Life](http://en.wikipedia.org/wiki/Benevolent_Dictator_for_Life) ([http://en.wikipedia.org/wiki/Benevolent\\_Dictator\\_for\\_Life](http://en.wikipedia.org/wiki/Benevolent_Dictator_for_Life)). Guido still signs off on all major changes to the core Python language.

## Variables In-class Exercise

- Spent 5 min max:
  1. Comment every line of code sensibly
  2. Create two variables assignments
  3. One with your Name
  4. One with Surname
  5. And greet yourself using a print function

```
In [20]: # Save name, as text string, into a variable
```

```
In [21]: # Save surname, as text string, into a variable
```

```
In [22]: # Print user greeting, using print function, and using + concatenation operator to glue strings together
```

[\[Back to top\]](#)

## 8. Strings

Strings are sets of characters. Strings are easier to understand by looking at some examples.

### 8.1 Single and double quotes

Strings are contained by either single or double quotes.

```
In [23]: my_string = "This is a double-quoted string."
my_string = 'This is a single-quoted string.'
```

This lets us make strings that contain quotations.

```
In [24]: quote = "Linus Torvalds once said, 'Any program is only as good as it is useful.'"
```

### 8.2 Changing case

You can easily change the case of a string, to present it the way you want it to look.

```
In [25]: first_name = 'eric'

print(first_name)
print(first_name.title())
```

```
eric
Eric
```

It is often good to store data in lower case, and then change the case as you want to for presentation. This catches some TYpos. It also makes sure that 'eric', 'Eric', and 'ERIC' are not considered three different people.

Some of the most common cases are lower, title, and upper.

```
In [26]: first_name = 'eric'
```

```
print(first_name)
```

```
eric
```

```
In [27]: print(first_name.title())
```

```
Eric
```

```
In [28]: print(first_name.upper())
```

```
ERIC
```

```
In [29]: first_name = 'Eric'  
print(first_name.lower())
```

```
eric
```

In Python, you will see the above syntax quite often, where a variable name is followed by a dot and then the name of an action, followed by a set of parentheses. The parentheses may be empty, or they may contain some values.

```
variable_name.action()
```

In this example, the word "action" is the name of a method. A method is something (like a function or macro) that can be done to a variable. The methods 'lower', 'title', and 'upper' are all functions that have been written into the Python language, which do something to strings. Later on, you will learn to write your own methods.

[\[Back to top\]](#)

## 9. Numbers, String and Boolean Types

The following code gives us a brief introduction to the different data types.

They mainly include:

- Integers
- Floating point numbers
- String and Boolean Types

### 9.1 Integers and arithmetic

Integer numbers refer to any number that is a **whole number** and does not contain a decimal or fraction part.

You can **do all of the basic operations with integers**, and everything **should behave as you expect**.

Addition and subtraction use the standard plus and minus symbols.

Multiplication uses the asterisk, and division uses a forward slash.

Symbol	Task Performed
+	Addition
-	Subtraction
/	division
%	mod
*	multiplication
//	floor division
**	to the power of

```
In [30]: print(3+2)
```

```
5
```

```
In [31]: print(3-2)
```

```
1
```

```
In [32]: print(3*2)
```

```
6
```

```
In [33]: print(3/2)
```

```
1.5
```

### 9.2 Floating-Point numbers

Floating-point numbers refer to any **number with a decimal point**. Just like rands and cents.

```
In [34]: print(70.10+0.75)
```

```
70.85
```

### 9.3 Convert an object to a given type:

We also call this type conversion.

With this, you can convert a given data type, such as a decimal number (integer), into a floating point number.

Or you can convert a number into string, a string which is a sequence of characters encapsulated with double or single quotes.

```
In [35]: float(2)
```

```
Out[35]: 2.0
```

```
In [36]: int(2.9)
```

```
Out[36]: 2
```

```
In [37]: str(2.9)
```

```
Out[37]: '2.9'
```

Let's construct an example to show maybe one example of where the above conversion functions can help us.

Say we have a variable that contains a string (like text), but the text contains a number:

```
In [38]: number_one = '500'
```

If we want to add a whole number (integer) to it, Python will error:

```
In [39]: print(number_one + 200)
```

```
-----  
TypeError                                 Traceback (most recent call last)  
<ipython-input-39-e839c8c45f8c> in <module>  
----> 1 print(number_one + 200)
```

```
TypeError: can only concatenate str (not "int") to str
```

We can solve this problem by converting the string "500" to a number, using the *int* function:

```
In [40]: print(int(number_one) + 200)
```

```
700
```

Just bare in mind that these **type conversion** functions **will return a result**, the result can ideally be **saved into a new variable** name for later use.

```
In [41]: # Save the returned data type into a new variable  
cost = float("749.99")
```

```
In [42]: cost + 0.01
```

```
Out[42]: 750.0
```

### 9.4 Rounding numbers

*round( )* function rounds the input value to a specified number of places or to the nearest integer.

```
In [43]: # round(number, ndigits=None)  
# If no ndigits parameter is specified, it will return a integer (whole number)  
  
print (round(5.6231))  
print (round(4.55892, 2))
```

```
6  
4.56
```

### 9.5 Generating a range of integer numbers

*range( )* function outputs the integers of the specified range.

It can also be used to generate a series by specifying the difference between the two numbers within a particular range.

The elements are returned in a Python list useable in other Python constructs (will be discussing lists in detail later.)

To visually "see" what a range function actually generates, let's convert the ranges above into a proper Python list using the built-in 'list' conversion function:

```
In [44]: print(list(range(1,15)))  
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
```

## Numbers, String and Boolean Types In-Class Exercise

- Spent 5 min max:
  1. Comment every line of code sensibly
  2. Assign an integer "2" to a variable called "my\_int"
  3. Round off the floating-point number 8.7756 and store it into a variable called "my\_num"
  4. Multiply the two variable and assign the answer to a variable called "answer"
  5. Convert the integer or whole number answer stored in "answer", to a floating-point number
  6. Print out the variable using the print function

```
In [45]: # Assign an integer "2" to a variable called "my_int"
```

```
In [46]: # Round off the floating-point number 8.7756 and store it into a variable called "my_num"
```

```
In [47]: # Multiply the two variable and assign the answer to a variable called "answer"
```

```
In [48]: # Convert the integer or whole number answer stored in "answer", to a floating-point number
```

```
In [49]: # Print out the variable using the print function
```

[\[Back to top\]](#)

## 10. Collections of data - Lists and Dictionaries

Let's have a look at working with collections of data in Python.

This could for instance be collections of data in lists, or collections into dictionaries.

Let's have a look at the **difference between lists and dictionaries**.

### 10.1 Lists

In simple terms, data structure are the the collection or group of data in a particular structure.

**List are collections of items.**

These items can be anything you want:

- Strings
- Integers
- Floating-Point Numbers

Lists are the most commonly used data structure.

Think of it as a sequence of data that is enclosed in square brackets and data are separated by a comma.

Each of these data can be accessed by calling it's index value.

**Lists are declared** or defined by just equating a variable to '[' ] or list.

```
In [50]: # This is an empty list  
a = []
```

One can directly assign the sequence of data to a list x as shown.

```
In [51]: my_list = ['apple', 'orange', 777]
```

```
Out[51]: ['apple', 'orange', 777]
```

### 10.1.1 Indexing

In Python, Indexing starts from 0.

We call this lists are zero-indexed.

Index	Value
0	apple
1	orange
2	777

The list x, which has three elements, will have apple at 0 index and orange at 1 index, and the number 777 at index 2.

```
In [52]: my_list[0]
```

```
Out[52]: 'apple'
```

### 10.1.2 Slicing

Indexing was only limited to accessing a single element.

Slicing on the other hand is accessing a sequence of data inside the list.

In other words "slicing" the list.

Slicing is done by defining the index values of the first element and the last element from the parent list that is required in the sliced list. It is written as parentlist[ a : b ] where a,b are the index values from the parent list. If a or b is not defined then the index value is considered to be the first value for a if a is not defined and the last value for b when b is not defined.

```
In [53]: num = [5,6,7,8,9,10,11]
```

```
num
```

```
Out[53]: [5, 6, 7, 8, 9, 10, 11]
```

Let's now print certain items from in Python list called *num*.

We use slicing to only display **from** the very first item (index number **0**), and we'll **slice 3 items**:

```
In [54]: print (num[0:3])
```

```
[5, 6, 7]
```

Let's slice to only display from index number 4 (the 5th element), and we'll slice all the way to the end of the list:

```
In [55]: print (num[4:])
```

```
[9, 10, 11]
```

### 10.1.3 Built in List Functions

To find the length of the list or the number of elements in a list, **len( )** is used.

```
In [56]: len(num)
```

```
Out[56]: 7
```

If the list consists of all integer elements then **min( )** and **max( )** gives the minimum and maximum value in the list.

```
In [57]: min(num)
```

```
Out[57]: 5
```

```
In [58]: max(num)
```

```
Out[58]: 11
```

`append()` is used to add a element at the end of the list.

```
In [59]: lst = [1,1,4,8,7]
```

```
In [60]: lst.append(1)
print (lst)

[1, 1, 4, 8, 7, 1]
```

Python offers built in operation `sort()` to arrange the elements in ascending order.

For descending order, By default the reverse condition will be False for reverse. Hence changing it to True would arrange the elements in descending order.

```
In [61]: lst.sort(reverse=True)
print (lst)

[8, 7, 4, 1, 1, 1]
```

Similarly for lists containing string elements, `sort()` would sort the elements based on its ASCII value in ascending and by specifying reverse=True in descending.

```
In [62]: names = ['Earth', 'Air', 'Water', 'Fire']
```

```
In [63]: names.sort()
print (names)

['Air', 'Earth', 'Fire', 'Water']
```

```
In [64]: names.sort(reverse=True)
print (names)

['Water', 'Fire', 'Earth', 'Air']
```

## 10.2 Dictionaries

Dictionaries are also collections of items.

Dictionaries are more used like a database because here you can index a particular sequence with your user defined string.

These are still groups of data, but instead of numeric zero-based indexes like lists, they use key / value pairs.

In other words, we have to define the index/key when creating the dictionary.

The index or key can be any valid data type:

- Integer
- String
- Floating-Point Number

Dictionaries are declared or defined by just equating a variable to '{ }'.

Notice how the key are defined before the : (colon), and the value after the colon.

New items are seperated by commas:

```
In [65]: my_dictionary = {'First': "apple", 'Favourite': "orange", 4.0: 777}
my_dictionary
```

```
Out[65]: {'First': 'apple', 'Favourite': 'orange', 4.0: 777}
```

Visually a dictionary key/value pairs can be presented as follows:

Index/Key	Value
"First"	apple
"Favourite"	orange
4.0	777

```
In [66]: my_dictionary['First']
```

```
Out[66]: 'apple'
```

## Lists In-Class Exercise

- Spent 5 min max:
  1. Comment every line of code sensibly
  2. Create a list containing 5 holiday destination name
  3. Sort the list using the .sort method
  4. Print out the sorted list using the print function

```
In [67]: # Create a list containing 5 holiday destination name
```

```
In [68]: # Sort the list using the .sort method
```

```
In [69]: # Print out the sorted list using the print function
```

[\[Back to top\]](#)

## 11. User-define Functions

Quite often, in an algorithm, the statements and lines of code keep repeating itself.

It will be a tedious job to execute the same statements again and again and will consume a lot of memory and is not efficient.

Functions helps us solve this problem.

Functions are in some way like macros, just for the purpose of analogy.

---

This is the basic syntax of a function

```
def funcname(argument1, arg2,... argN):  
    statements  
    return <value>
```

Read the above syntax as:

- A function by name "funcname" is defined
  - Which accepts arguments "argument1,arg2,...argN".
  - The function after executing the statements returns a "value".
- 

**Let's consider a simple example.**

Perhaps we might require the next two lines over and over later on in our program or code:

```
In [70]: print ("Hey Rajath!")  
print ("Rajath, How do you do?")
```

```
Hey Rajath!  
Rajath, How do you do?
```

Instead of writing the above two statements every single time, it can be **replaced by defining a function** which would **do the job** in just one line.

```
In [71]: # Defining a function called GreetRajath()  
def GreetRajath():  
    print ("Hey Rajath!")  
    print ("Rajath, How do you do?")
```

Now, whenever we need those two lines of code, we can just **call upon our function** to execute them for us, it will **call those two print statements**:

```
In [72]: GreetRajath()
```

```
Hey Rajath!  
Rajath, How do you do?
```

One limitation we now have, is that `GreetRajath()` prints the message for a single person's name, it's always "Rajath".

To improve on this, we can make our function `GreetUser()` accept arguments which will store the name and then prints the respective name.

To do so, add a argument within the function can be added.

The argument we added here under are the "username" in: `GreetUser(username)`

In the below function, "username" will become a temporary variable inside the function's algorithm (code).

```
In [73]: def GreetUser(username):  
    print("Hey", username + '!')  
    print("Python is a fun programming language")
```

We can now call the above function, but pass an argument called "username", for use inside the function.

```
In [74]: GreetUser("Philip")
```

```
Hey Philip!  
Python is a fun programming language
```

We are also able pass a **string variable**, say `the_username` to the function:

```
In [75]: the_username = "Pi Guy"
```

As mentioned earlier, this argument will now also become a variable in the function code.

In other words, "the\_username" is passed as "username" in the function.

```
In [76]: GreetUser(the_username)
```

```
Hey Pi Guy!  
Python is a fun programming language
```

Let's look at **another extended functions example** which explains how you can define your own **user defined function**:

Let's create a user defined function which we'll call "calculate\_PAYE", we'll pass it three parameters/arguments:

- salary (as a number)
- name (as text)
- surname (as text)

This function's objective should be to: calculate the PAYE and print out how much it is.

```
In [77]: def calculate_PAYE(salary, name, surname):  
    # Calc PAYE tax as 30% of salary  
    paye = salary * 0.30  
    # Format PAYE as a floating point number with 2 decimals and commas as thousands separator  
    paye_formatted = "{:.2f}".format(paye)  
    # Now print out the calculated salary and PAYE  
    print(name, surname + "'s" + " salary is: R", salary, "and PAYE was calculated as R", paye_formatted )  
  
print("This code line was not 'TAB indented', and thus not part of the function definition")
```

This code line was not 'TAB indented', and thus not part of the function definition

```
In [78]: # Setup a a variable called salary  
salary=10000  
# Now call the function by passing this variable and name/surname values  
calculate_PAYE(salary, "Michelle", "Ramphomane")
```

Michelle Ramphomane's salary is: R 10000 and PAYE was calculated as R 3,000.00

```
In [79]: # Provide all function arguments as values in the function call:  
calculate_PAYE(7000, "Danny", "Kavishe")
```

Danny Kavishe's salary is: R 7000 and PAYE was calculated as R 2,100.00

[\[Back to top\]](#)

## 12. Control Flow Statements

### 12.1 Indentation

Leading whitespace (spaces and tabs) at the beginning of a logical line is used to compute the indentation level of the line, which in turn is used to determine the grouping of statements.

Tabs are replaced (from left to right) by one to eight spaces such that the total number of characters up to and including the replacement is a multiple of eight (this is intended to be the same rule as used by Unix). The total number of spaces preceding the first non-blank character then determines the line's indentation.

Indentation is rejected as inconsistent if a source file mixes tabs and spaces in a way that makes the meaning dependent on the worth of a tab in spaces; a TabError is raised in that case.

### 12.2 Relational Operators

Symbol	Task Performed
==	True, if it is equal
!=	True, if not equal to
<	less than
>	greater than
<=	less than or equal to
>=	greater than or equal to

```
In [80]: z = 1
```

```
In [81]: z == 1
```

Out[81]: True

```
In [82]: z > 1
```

Out[82]: False

### 12.3 If

```
if some_condition:
```

```
    algorithm
```

We set a variable "x" equal to the integer (or whole number) value of 12

We then test "some\_condition", in our case below, whether x is greater than 10.

Important to note how Python expects a colon (:) to show where the "conditional test" ends.

If the "conditional test" is True, then we execute or **run the code which are indented** directly under the "if" statement.

```
In [83]: x = 12  
if x > 10:  
    print("Hello, this code line ran because 'if' statement's test condition was 'True'")  
print("\nThis line is not part of the 'if' statement, will print regardless of 'if' statement")
```

Hello, this code line ran because 'if' statement's test condition was 'True'  
This line is not part of the 'if' statement, will print regardless of 'if' statement

### 12.4 If-else

```
if some_condition:
```

```
algorithm
```

```
else:
```

```
    algorithm
```

```
In [84]: x = 12
if x > 10:
    print ("x was greater than 10")
else:
    print ("x was NOT greater than 10")
```

```
x was greater than 10
```

## 12.5 if-elif-else

```
if some_condition:
```

```
    algorithm
```

```
elif some_condition:
```

```
    algorithm
```

```
else:
```

```
    algorithm
```

```
In [85]: x = 10
y = 12
if x > y:
    print ("x>y")
elif x < y:
    print ("x<y")
else:
    print ("x=y")
```

```
x<y
```

if statement inside a if statement or if-elif or if-else are called as nested if statements.

```
In [86]: x = 10
y = 12
if x > y:
    print ("x>y")
elif x < y:
    print ("x<y")
    if x==10:
        print ("x=10")
    else:
        print ("invalid")
else:
    print ("x=y")
```

```
x<y
x=10
```

## 12.6 For Loop

The syntax is:

for variable in something:

```
    algorithm #(Do something)
```

We effectively can get the "for" loop to iterate over every element in a **list**, and "**do something**" with **each element**.

As the "for" loop iterates over each element, it stores the next element into a temporary variable, "i" in this example:

```
In [87]: for i in [9, 8, 7, 6, "Pi" , "Guy", 1, 2.0]:
    print (i)
```

```
9
8
7
6
Pi
Guy
1
2.0
```

In the above example, `i` iterates over the 0,1,2,3,4.

Recall how the `range(n)` function generates a list of "n" numbers, starting with "0" by default.

By explicitly asking Python to represent the `range(7)` function as a list of integers, we see exactly this:

```
In [88]: list(range(7))
```

```
Out[88]: [0, 1, 2, 3, 4, 5, 6]
```

```
In [89]: for i in range(7):
    print (i)
```

```
0
1
2
3
4
5
6
```

## 12.7 While Loop

```
while some_condition:
    algorithm
```

With a while loop, a condition gets tested. We call this the "conditional test".

While the "conditional test" proves to be True (and not False), the while loop keep iterating the code one more time.

With every iteration of the while loop, it executes the algorithm or code inside the loop, the "inside the loop" part being denoted by all the code indented as here under:

```
In [90]: i = 1
while i < 4:
    print("The current value of variable i = ", i)
    i = i+1
print('Bye')
```

```
The current value of variable i =  1
The current value of variable i =  2
The current value of variable i =  3
Bye
```

Notice above how once the "conditional test" is: `i < 4` ?

Thus: is the value of variable "`i`" still less than the integer value 4, and if so, execute the code in the while loop once more.

The value of variable "`i`", however, gets increased with 1 each time the code gets executes (`i = i + 1`).

Once the value of "`i`" is 4, then 4 (`i=4`) is NOT less than 4 anymore, so the "conditional test" returns False, and the while loop ends itself and jumps to the first line of code AFTER the while loop's indented code.

---

Let's look at another **good example** which explains how a **while loop** works:

```
In [91]: # Load the time library's sleep function, so we can pause script execution
from time import sleep

# Set up a variable with initial value 1
my_num = 1

# While loop, but
while my_num < 6:
    # Let's add a print statement, which will print the iteration number for us
    print("This is iteration: ", my_num)

    # Let's say we switch ON a blue light, and wait 1 second (blue light on for 1 second)
    print("Blue On")
    #blue.on()
    sleep(1)

    # Let's say we switch OFF a blue light, and wait 1 second (blue light off for 1 second)
    print("Blue Off")
    #blue.off()
    sleep(1)

    # We increment (increase) the my_num variable's value with "1"
    my_num = my_num + 1

    # Now the while loop iteration ends here and
    # Python will go back up and check the conditional test

print("The blue light blinked 5 times, we are done now.")
```

```
This is iteration: 1
Blue On
Blue Off
This is iteration: 2
Blue On
Blue Off
This is iteration: 3
Blue On
Blue Off
This is iteration: 4
Blue On
Blue Off
This is iteration: 5
Blue On
Blue Off
The blue light blinked 5 times, we are done now.
```

[\[Back to top\]](#)

## B. Homework

This is your homework, a little bit of coding practice for the **next session**.

We will once again write a small **quiz/test** at the beginning of the **next session**.

### 1. Review this Jupyter Notebook

Estimated Time: Approx 15-20 minutes

Briefly review this Jupyter Notebook - Intro to Python Basics.

Also look at the available summary for this session.

### 2. Watch the short introduction video on Python Pandas

Pandas is an open-source python library that provides easy to use, high-performance data structures and data analysis tools.

During the next session, we will be looking at Pandas in more depth.

**Watch the following short 5 min video on Pandas:**

<https://www.youtube.com/watch?v=XDAnFZqJDvI> (<https://www.youtube.com/watch?v=XDAnFZqJDvI>)

If you do not have access to YouTube, you can always get the video from here:

<https://www.piguy.co.za/jupyter/pandas.mp4> (<https://www.piguy.co.za/jupyter/pandas.mp4>)

### 3. Watch the short video on xlwings and what it offers

Estimated Time: Approx 3 minutes

Visit the xlwings home page.

During the next session, we will be looking at xlwings to help interact with Excel in more depth.

xlwings - Python For Excel. Free & Open Source: <https://www.xlwings.org/> (<https://www.xlwings.org/>)

Here is a direct link for the same video, on YouTube: <https://www.youtube.com/watch?v=A3jrUXNokYk> (<https://www.youtube.com/watch?v=A3jrUXNokYk>)

**Watch this 3 min video on xlwings "INTERACTION / AUTOMATION"**

- It is this 3 min video on the **home page**, first one on the left if you scroll down: "xlwings: Excel automation / interaction"
- Description reads: Leverage Python's scientific stack for interactive data analysis using Jupyter Notebooks, NumPy, Pandas, scikit-learn etc. Or use xlwings to automate Excel reports with Python

### 4. Conditional Statement (While loop) Excercise

For this part of your homework, **write a while loop** in Python:

Control Flow Statements Excercise (See section 12.7)

- Spent 10 min max:
  1. Comment every line of code sensibly
  2. Set a tea\_temp variable equal to integer 76
  3. Write a while loop, which checks whether your tea's temperature is less than 54 degrees
  4. If it is still not less than 54, print out "add one more ice cube"
  5. At the end of while statement decrement tea\_temp variable's value with 1
  6. As soon as your tea's temperature is less than 54, exit while loop
  7. Print out, "Your tea is ready!"

### 5. User-defined Functions Excercise

User-defined Functions Excercise (See examples in section 11 to help you with this)

- Spent 10 min max:
  1. Comment every line of code sensibly
  2. Write your own **user defined function**
  3. Name your function **greetings**
  4. It should be **passed two arguments**, both being strings, one being "**name**" and another "**surname**"
  5. The function should print out: "Greeting 'your\_name' 'your\_surname', hope you are doing great"

[\[Back to top\]](#)

- END -