Python 101

# Session 01

## **What is Python?**

Python is a high level language developed by Guido van Rossum in the 80ties. It is considered as a modern language that is easy to read. This results in an easy learning curve and it has contributed to the popularity of Python.

## **Installation**

There are a few different options for the installation of python. First of all, we have to choose the version and second, we have to choose the programming environment. We will use python in version 3.x as opposed to version 2.7.

As for the programming environment, we will be using the IDE (Integrated Development Environment) that ships with the download of python. Obviously, there are other environments such as Pycharm, Jupyter Notebooks, etc - there is nothing wrong with those, feel free to use them if you wish to do so. In any case I will be using the IDLE for it is an easy common ground for everybody and perfect if one is new to programming.

The installation under windows is simple: Go to the webpage of python and download the latest version. During installation you will be asked where to install it and if the installer has the add a PATH variable, please make sure that you check the box and continue with the installation.

The installation under mac is the same, he just doesn’t ask you for the PATH variable and you click through the menu.

## **Basics**

### The Interactive Shell

Python runs in an interactive shell, where commands can be typed in and they are evaluated in real time. This behaviour is different from what we have seen from Arduino or Processing: You had to press run, the computer translated that into machine code and the program was running. The technical term is that python is an Interpreted language that runs on the back of an interpreter.

You can start to write longer programs if you open a new window under “File” > “New File”. Note that the menu bar changes and you will have now the option to run the program in the instance of the shell that runs.

### Variables and Data Types (Integer, Floating Point, Strings, simple array)

Unlike Arduino (c language) or processing (java), Python does not need to know the type of the variable, a declaration like this is is correct:

|  |
| --- |
| >>> a = 4 >>> a 4 |

The variable “a” can also contain a string, a float or an array. The last line counts.

|  |
| --- |
| >>> a = "Hello You" >>> a 'Hello You' >>> a=[5,6,4] >>> a [5, 6, 4] |

This behaviour of python is also characterized as “**weakly typed** (**loosely typed**)”.

If you are not sure what a certain variable is, this helps you:

|  |
| --- |
| >>> type(a) <**class** '**list**'> |

The shell can also work as a calculator:

|  |
| --- |
| >>> 5+4 9 >>> 5+4\*2 13 >>> 4/5 0.8 >>> 2/3 0.6666666666666666 >>> w=4 >>> w+5 9 |

The most common data types are integer, floats and strings. They can be combined into arrays as such:

|  |
| --- |
| >>> a= [2, 6.7, "Hello"] >>> a [2, 6.7, 'Hello'] |

Sometimes, it’s necessary to transform one data type into another, like an integer to a float. This is called ***casting*** and works like that:

|  |
| --- |
| >>> float(9) 9.0 >>> int(4.68343) 4 >>> str(4) '4' |

Things like that don’t work - this gives an error message.

|  |
| --- |
| >>> int("Anna") |

Strings have some interesting behaviors:

|  |
| --- |
| >>> "Car" + "Bus" 'CarBus' |

> this is called ***string concatenation***

|  |
| --- |
| >>> "Bus " \* 3 'Bus Bus Bus ' |

### Comments

Commenting out means that the code will be ignored. A single line:

|  |
| --- |
| # This is a comment |

And over many lines:

|  |
| --- |
| """ This is a comment Over many lines """ |

### Escape Characters

Strings have a few special characters that allows us to some simple formatting, they are called escape characters:

|  |
| --- |
| \n **is** a new line \t **is** a tab \\ **is** a backslash |

A code sample would look like:

|  |
| --- |
| >>> a = "This is the escape character \n for a new line" >>> b =" This is the escape character \t for a tab" >>> c = "This is a \\ backslash" >>> **print** (a) This **is** the escape character   **for** a new line >>> **print** (b)  This **is** the escape character **for** a tab >>> **print** (c) This **is** a \ backslash |

Now, the difficulty lies when you have to use the combination “\n” or “\t” for any reason and you don’t want to have the test formatted. The r in front of the script will treat the string as raw text:

|  |
| --- |
| >>> w = "C:\Users\vince\Dropbox\new\test" SyntaxError: (unicode error) 'unicodeescape' codec can't decode bytes in position 2-3: truncated \UXXXXXXXX escape >>> w = r"C:\Users\vince\Dropbox\new\test" >>> print (w) C:\Users\vince\Dropbox\Office Docs\Library |

Please consider this when you work with paths and filenames.

As an alternative you can define a file path like that as well:

|  |
| --- |
| w = "C:\\Users\\vince\\Dropbox\\new\\test" |

### First Program

Now, let’s put all the above into action:

|  |
| --- |
| # a very polite script print("Hello") name = input("What is your name?") print("Nice to meet you " + name) **print** ("Have a nice day!") |

### Running scripts in the command line

As an alternative to the shell, you can run scripts directly in the command line of windows. To do so, navigate to the folder where the script is and type “cmd” into the address bar. The command line interface appears and the working directory is set automatically to the location of the explorer window. Not type the name of your script and see how it runs in the command line.

## Flow Control

The logic of python’s flow control is similar to the one of arduino, also python knows booleans, if statements and so forth.

### Boolean Values and Operators

True and False are the boolean values that control the flow of a programm. They can be typed in or they are the result of a certain expression:

|  |
| --- |
| >>> 3>1 **True** >>> 3+3==6 **True** >>> "Anna" == "Anna" **True** >>> "Anna" == 4 **False** >>> e=3 >>> r=5 >>> e==r **False** |

Or you can simply do:

|  |
| --- |
| >>> x=**True** |

These can be combined with AND or OR expression following these rules:

|  |
| --- |
| >>> **True** **True** >>> **False** **False** >>> **True** **and** **True** **True** >>> **True** **and** **False** **False** >>> **False** **and** **False** **False** >>> **True** **or** **True** **True** >>> **True** **or** **False** **True** >>> **False** **or** **False** **False** |

### In and not operators in strings

### If statements

|  |
| --- |
| >>> **if** 6>3:  print("yes this is correct")  yes this **is** correct |

Note the special way python deals with a code block, unlike c based languages, there is an indentation and a colon. A same code in processing would be:

|  |
| --- |
| **if** (3>1){  println("Yes, correct");  } |

Note that the indentation and the whitespace play a very important role in python.

### Else statements

|  |
| --- |
| **if** 6>9:  print("yes, this is correct") **else**:  print("Nope") |

### Elif statements

|  |
| --- |
| **if** **False**:  print("First test is true") **elif** **False**:  print("Second test is true") **elif** **False**:  print("Third test is true") **else**:  print("All test conditions are False") |

### For loops and the range function

The range function evaluates to a series of numbers. This expression is evaluated in certain circumstances. This means that if you simply type

|  |
| --- |
| >>> range(10,20) |

The result is

|  |
| --- |
| range(10, 20) |

This function is typically used in a ***for loop***:

|  |
| --- |
| >>> **for** i **in** range (10,20):  print(i)   10 11 12 13 14 15 16 17 18 19 |

Explore different options of the range function:

|  |
| --- |
| **for** i **in** range (20):  print(i) **for** i **in** range (2,20,4):  print(i) **for** i **in** range (20,2,-1):  print(i) |

For Loops are also allowed to run in lists directly like:

|  |
| --- |
| >>> **for** i **in** [3,5,80,"Car",[4,5,6]]:  print(i)  3 5 80 Car [4, 5, 6] |

Considering that “i” can be replaced by any other expression, one can write:

|  |
| --- |
| brands = ["Coca Cola", "Fanta", "7Up", "Sprite"]  **for** brand **in** brands:  **print** (brand) |

### While loops

While loops are usually used to create an loop that is only exited when a certain condition is met:

|  |
| --- |
| print('Please type your name.') name = input() **while** name != 'peter':  print('Please type your name.')  name = input() print('Thank you!') |

The above is the most common usage of a while loop. But it can also be used as a replacement for a for loop in some cases:

|  |
| --- |
| index = 0 **while** car < 5:  print('Hello, world.')  index = index + 1 |

## Functions

Functions allow you structure your code into modules. Parts that are often used can be taken out of the main code and declared as a separate piece of code at the beginning of the script.

|  |
| --- |
| **def** **fib**(n): # write Fibonacci series up to n  """Print a Fibonacci series up to n."""  a, b = 0, 1  #short for  #a = 0  #b= 1  **while** a < n:  print(a, end=' ')  a, b = b, a+b  print()  fib(2000) |

Functions are located at the beginning of the script. The value in between the parentheses is called argument and when you pass a value is it’s called parameter.

The above function simply prints values into the command line, alternatively, we could also include a return statement that returns a value. In this case we return a value

|  |
| --- |
| **def** **fib**(n): # write Fibonacci series up to n  """Print a Fibonacci series up to n."""  a, b = 0, 1  #short for  #a = 0  #b= 1  **while** a < n:  print(a, end=' ')  a, b = b, a+b  print("finished")  **return** a   fib(2000)  x = fib(2000) |

## Keywords Arguments in print()

We have been using the print() function in previous scripts. The regular print function looks like

|  |
| --- |
| **print** ("This is", end = ' ') **print** ("the way") |

But Python allows you to use Keywords in certain cases. Keywords are extra option that come in a format like *[name] = [value]*

|  |
| --- |
| **print** ("This is", end = ' ') **print** ("the way")  **print** ("this ", "is ", "the ", "way ", sep = '\*') |

## **Lists**

Lists in python hold an array of values. As we have seen, lists can contain different values and the length can be dynamically adjusted. This makes lists flexible and easy to work with.

|  |
| --- |
| myList = [ "Car", "Bus", "Ship"] |

Items can be accessed with this statement:

|  |
| --- |
| myList[0] |

Or

|  |
| --- |
| myList[2] |

Values in ca list can be changed with this method:

|  |
| --- |
| myList[0]= "Airplane" |

To add a value

|  |
| --- |
| myList.append(3) |

If we want to get specific parts from a list, like a sublist, it’s called *list slicing*. Let’s do a new list:

|  |
| --- |
| >>> myList = [ "Car", "Bus", "Ship", "Airplane", "Tube", "Train"] >>> myList[2:4] ['Ship', 'Airplane'] |

You can also leave things out like this:

|  |
| --- |
| >>> myList[:2] ['Car', 'Bus'] |

You can also count backwards:

|  |
| --- |
| >>> myList[-1] 'Train' >>> myList[-2] 'Tube' |

You can remove elements:

|  |
| --- |
| >>> **del** myList[2] >>> myList ['Car', 'Bus', 'Airplane', 'Tube', 'Train'] |

And list can have many dimensions, like an array of RGB pixels:

|  |
| --- |
| >>> image = [[230,10,40],[23,78,67],[56,78,123]] >>> image[1] [23, 78, 67] >>> image [1][0] You can join two lists >>> newList = myList + image >>> newList ['Car', 'Bus', 'Airplane', 'Tube', 'Train', [230, 10, 40], [23, 78, 67], [56, 78, 123]] |

You can also zip lists like this:

|  |
| --- |
| >>> zipped = zip(myList, image) >>> **for** name **in** zipped:  print(name) ('Car', [230, 10, 40]) ('Bus', [23, 78, 67]) ('Airplane', [56, 78, 123]) |

Here we have “zipped” two lists in pairs. This might be of interest when you want to combine x and y values of a point.

## Tuples

Tuples are the same as lists, but they are immutable. Like strings, they can not be changed. A tuple can be like:

|  |
| --- |
| >>> myTuple = (2,5,3,6) >>> myTuple (2, 5, 3, 6) |

I rarely use them and they an easily converted into a list like that:

|  |
| --- |
| >>> list(myTuple) [2, 5, 3, 6] |

## **Dictionaries**

Python knows a data type that works on a *key : value* basis. It’s called Dictionary and looks like:

|  |
| --- |
| >>> dict = {  "brand": "Ford",  "model": "Mustang",  "year": 1964 }  >>> dict {'brand': 'Ford', 'model': 'Mustang', 'year': 1964} |

You can access the values like so

|  |
| --- |
| >>> print(dict["brand"]) Ford |

Changing values goes goes like this:

|  |
| --- |
| >>> dict["brand"] = 'bmW' >>> dict {'brand': 'bmW', 'model': 'Mustang', 'year': 1964} |

Dictionaries do have these rules:

* They don’t have an order. This is unlike lists where values stay in their place
* Key’s can’t be duplicated. In the example above, an entry with two ‘brands’ is not allowed.
* Dictionaries can change, pairs can be changed, removed or added

### Typical methods

Get the value of a key

|  |
| --- |
| >>> x = dict["model"] >>> x 'Mustang' |

or

|  |
| --- |
| >>> x = dict.get("model") >>> x 'Mustang' |

Get all the keys of a dictionary

|  |
| --- |
| >>> x = dict.keys() >>> x dict\_keys(['brand', 'model', 'year']) |

This gets all the values as a list

|  |
| --- |
| >>> x = dict.values() >>> x dict\_values(['Ford', 'Mustang', 1964]) |

Here you get the pairs in a list as tuple items

|  |
| --- |
| >>> x = dict.items() >>> x dict\_items([('brand', 'Ford'), ('model', 'Mustang'), ('year', 1964)]) |

You can check if a key exists

|  |
| --- |
| >>> **if** "model" **in** dict:  print("Yes, 'model' is one of the keys in the thisdict dictionary")    Yes, 'model' **is** one of the keys **in** the thisdict dictionary |

Add items

|  |
| --- |
| >>> dict["color"] = "red" >>> dict {'brand': 'Ford', 'model': 'Mustang', 'year': 1964, 'color': 'red'} |

If you are not sure whether the key already exists or not, you can update the dictionary

|  |
| --- |
| >>> dict.update({"year": 2020}) >>> dict.update({"owner": "john"}) >>> dict {'brand': 'Ford', 'model': 'Mustang', 'year': 2020, 'color': 'red', 'owner': 'john'} |

Or you remove items

|  |
| --- |
| >>> dict.pop("model") 'Mustang' >>> dict {'brand': 'Ford', 'year': 2020, 'color': 'red', 'owner': 'john'} |

And here an overview of all commands:

clear() Removes all the elements from the dictionary  
copy() Returns a copy of the dictionary  
fromkeys() Returns a dictionary with the specified keys and value  
get() Returns the value of the specified key  
items() Returns a list containing a tuple for each key value pair  
keys() Returns a list containing the dictionary's keys  
pop() Removes the element with the specified key  
popitem() Removes the last inserted key-value pair  
setdefault() Returns the value of the specified key. If key does not exist: insert the key

update() Updates the dictionary with the specified key-value pairs  
values() Returns a list of all the values in the dictionary

## **Reading and writing files**

Reading and writing files is quite simple. You always create an object on which you apple commands to. Like “open”, “read” or “write” are commands onto the file object.

### Open Files and file paths

Python needs to know where the file is or where to save it. *Absolute file paths* have the full path:

|  |
| --- |
| C:\Users\vince\AppData\Local\Programs\Python\Python38 |

*Relative Paths* indicates folders in relation to the current directory. In this case, the exact location doesn’t really matter. The current directory is the folder where the python file is saved. If you place a file directly into the work directory, you can simply open it like this and without any extra path information.

|  |
| --- |
| file = open("Text.txt", "r") |

In case it’s a subfolder, you can define the location like:

|  |
| --- |
| file = open(".\\folder\\demofile.txt", "r") |

The parameter at the end indicates:

|  |
| --- |
| "r" - Read - Default value. Opens a file for reading, error if the file does not exist "a" - Append - Opens a file for appending, creates the file if it does not exist "w" - Write - Opens a file for writing, creates the file if it does not exist "x" - Create - Creates the specified file, returns an error if the file exists |

### Read/write files

Once you have the file open, you will need to tell python what to do with the file. In the case of the text or csv file you can read the entire file or you can read the file by line:

Read by lines:

|  |
| --- |
| file = open(".\\csv\\data.csv", "r") lines = file.readlines() **for** line **in** lines:  **print** (line) |

Read the entire text file without line breaks:

|  |
| --- |
| lines = file.read() |

The above will let you read files. If you want to write or append to a file, you will need to include the correct extension ad above.

|  |
| --- |
| file = open(".\\csv\\data.csv", "r") |

In this case, python will let us append a value to the end of the file. This is not unsimilar to the print () command, jus that you print to a file.

|  |
| --- |
| >>> baconFile = open('bacon.txt', 'w') >>> baconFile.write('Hello world!\n') 13 #this returns the number of characters written >>> baconFile.close() >>> baconFile = open('bacon.txt', 'a') >>> baconFile.write('Bacon is not a vegetable.') 25 >>> baconFile.close() >>> baconFile = open('bacon.txt') >>> content = baconFile.read() >>> baconFile.close() >>> print(content) |

After we have done that, we have to close the file. The above method is long:

|  |
| --- |
| file = open(".\\csv\\data.csv", "r") lines = file.readlines() **for** line **in** lines:  print(line) file.close |

And you don’t forget to close the file afterwards! In alternative there is a more compact way of writing the same:

|  |
| --- |
| **with** open('.\\csv\\data.csv', 'r') **as** file:  lines = file.readlines()  **for** line **in** lines:  print(line) |

The advantage of this one is that you save one line and that you make sure that the file is being closed.

## **Modules** and the **Python Standard library**

Python comes with a set of pre installed libraries that extend the functionality of python, they are called the standard libraries. The documentation is online:

<https://docs.python.org/3/library/>

If you want to see what is currently installed, type

|  |
| --- |
| help("modules") |

And if you want to see more details about a specific module, you can type:

|  |
| --- |
| help("math") |

### Math Module

Extended mathematical functions are not part of the python language, you will need to call a module of the standard library called “math”.

|  |
| --- |
| >>> **import** math **as** math >>> x = math.radians(90) >>> x 1.5707963267948966 |

### Pip Install

One of the most important modules is “pip install” - a library that lets you install third party libraries. Usually these libraries are saved in a central web base repository

<https://pypi.org/>

The installation is relatively simple. *Windows* open the command line and type in

|  |
| --- |
| pip install [package name] |

Like

|  |
| --- |
| pip install numpy |

or

|  |
| --- |
| pip install opencv-python |

Pip install under *Mac* is

|  |
| --- |
| sudo pip3 install [numpy **or** whatever the package name **is**] |

### The CSV module

The CSV File Reading and Writing module offers some extra features to read and write csv files. As usual, the library will be imported with

|  |
| --- |
| **import** csv |

And typically you would open a file with

|  |
| --- |
| **import** csv **with** open('.\\csv\\data.csv', newline='') **as** csvfile:  reader = csv.reader(csvfile,delimiter=',')  **for** row **in** reader:  print(', '.join(row)) |

Or - if you need the lines as lists you can simply do:

|  |
| --- |
| **import** csv **with** open('.\\csv\\data.csv', newline='') **as** csvfile:  reader = csv.reader(csvfile,delimiter=',')  **for** row **in** reader:  print(row) |

Writing a file goes like:

|  |
| --- |
| **import** csv  **with** open('values.csv', mode='w') **as** file:  writer = csv.writer(file, delimiter=',')   writer.writerow([30, 50, 3.55])  writer.writerow([40, 80, 34.44]) |

The example above gives you very little advantage above the conventional method, yet you might want to keep that in mind when you handle csv files and they need to have some format features that the usual command doesn’t have.

### Time Module

Time is notoriously difficult to calculate, the units are in 24 or 60, months and years have different length. And this is where the time module comes in. It lets you add and subtract days, months and hours. You can measure periods and format the date in many different ways.

Traditionally, scripting languages split into *times* and *dates*. Python does that as well, but has introduced a data type that combines both, the *datetime*. This is handy because you just deal with one data format - yet you can always use the other datatypes if you need to do so.

The display of today

|  |
| --- |
| >>> **import** datetime >>> x = datetime.datetime.now() >>> print(x) 2021-02-05 07:30:53.823842 |

The single elements of the date can be accessed like this:

|  |
| --- |
| >>> print(x.year) 2021 >>> print(x.day) 5 >>> print(x.second) 53 |

And if you want to set a date:

|  |
| --- |
| b = datetime.datetime(2020, 5, 17) |

And setting a date with hours, minutes, seconds and microseconds

|  |
| --- |
| c = datetime(2017, 11, 28, 23, 55, 59, 342380) |

As explained before, the way of calculating time is very tricky. It becomes even more difficult when you take into account different time zones and calendar types ( like how to convert dates between islamic calendar, gregorian calendar and Chinese calendar?) To mitigate this, a new way of measuring time has been established for computers: the epoch that counts the time from midnight January 1, 1970 onwards. It comes in formats like 1326244364. The conversion is as like:

|  |
| --- |
| >>> **from** datetime **import** date >>> timestamp = date.fromtimestamp(1326244364) >>> print("Date =", timestamp) Date = 2012-01-11 |

The output can be formatted in different ways:

|  |
| --- |
| >>> **from** datetime **import** datetime >>> now = datetime.now() >>> b = now.strftime("%d/%m/%Y, %H:%M:%S") >>> # dd/mm/YY H:M:S format >>> print(b) 06/02/2021, 08:56:29 |

Or just something like this if you want to get only one part of the date

|  |
| --- |
| >>> print(now.strftime("%B")) February |

You can access the shortcuts from here or here

<https://www.w3schools.com/python/python_datetime.asp>

<https://docs.python.org/3/library/time.html#module-time>

## Help and Resources

### Help

Python ships with an inbuilt help command.

|  |
| --- |
| >>> help (**print**) Help on built-**in** function **print** **in** module builtins:  print(...)  print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=**False**)    Prints the values to a stream, **or** to sys.stdout by default.  Optional keyword arguments:  file: a file-like object (stream); defaults to the current sys.stdout.  sep: string inserted between values, default a space.  end: string appended after the last value, default a newline.  flush: whether to forcibly flush the stream. |

More examples go ilke:

|  |
| --- |
| >>> help(open) |

Or if it’s a module you are not familiar with, you can do this:

|  |
| --- |
| >>> **import** math **as** math >>> dir(math) ['\_\_doc\_\_', '\_\_file\_\_', '\_\_loader\_\_', '\_\_name\_\_', '\_\_package\_\_', '\_\_spec\_\_', 'acos', 'acosh', 'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'comb', 'copysign', 'cos', 'cosh', 'degrees', 'dist', 'e', 'erf', 'erfc', 'exp', 'expm1', 'fabs', 'factorial', 'floor', 'fmod', 'frexp', 'fsum', 'gamma', 'gcd', 'hypot', 'inf', 'isclose', 'isfinite', 'isinf', 'isnan', 'isqrt', 'lcm', 'ldexp', 'lgamma', 'log', 'log10', 'log1p', 'log2', 'modf', 'nan', 'nextafter', 'perm', 'pi', 'pow', 'prod', 'radians', 'remainder', 'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'tau', 'trunc', 'ulp'] >>> help(math.sin) Help on built-**in** function sin **in** module math:  sin(x, /)  Return the sine of x (measured **in** radians). |

The example above has a few steps. First you import the module, then with dir() you show the continent of that directory and with help() you get more info regarding the command.

### Resources

There are plenty of resources online and as books. Most of the time I end up on these webpages:

* <https://www.w3schools.com/python/default.asp>
* <https://realpython.com/>
* <https://stackoverflow.com/>

Yet there are also larger tutorials such as

* <https://nbviewer.jupyter.org/github/Tanu-N-Prabhu/Python/blob/master/How_to_get_started_coding_in_Python%3F.ipynb>
* <https://wiki.python.org/moin/BeginnersGuide/Programmers>

As for specific books, there is one excellent reference here:

* Automate the Boring Stuff with Python, 2nd Edition: Practical Programming for Total Beginners (Al Sweigart)
* Introducing Python: Modern Computing in Simple Packages (Bill Lubanovic)

### Using Python

This tutorial give you a start to use python in few software packages:

* Rhino/Grasshopper: Rhino/Grasshopper works with Python 2.7, so you will need to consider slight changes in your code. Other than that, the online documentation is rather excellent.
* Houdini: houdini offers you to code in python
* Maya
* Blender
* Unity and Unreal
* … and countless libraries

### Implementations

After all this exciting news, a word of warning:

When people speak of Python, they often mean not just the language but also the *CPython implementation*. Python is actually a *specification for a language* that can be implemented in many different ways. The different implementations may be for compatibility with other libraries, or maybe for a little speed. Pure Python libraries should work regardless of your Python implementation, but those built on C (like NumPy) won’t. This section provides a quick rundown on the most popular implementations:

* CPython : CPython is the reference implementation of Python, written in C. It compiles Python code to intermediate bytecode which is then interpreted by a virtual machine. CPython provides the highest level of compatibility with Python packages and C extension modules.
* Jython: Jython is a Python interpreter implementation that compiles Python code to Java bytecode which is then executed by the JVM. If you use processing in python, it runs on jython.
* IronPython: IronPython is the python that rhino uses. It comes in python 2.7 and it looks like that it’s going to stay that way for a bit longer.
* MicroPython: MicoPython is for microcontrollers, similar to arduino.

This means that not all the third party libraries that work under CPython also work under the IronPython implementation of rhino. So you can not import Numpy and Scipy directly into grasshopper, you will need to take detours (https://wiki.mcneel.com/grasshopper/cpython)

### 