FIRST STEPS IN PYTHON

A good way to discover Python are the tutorials from Open Classroom (previously known as Site du zero, for the older ones…). Instead of redoing it all myself, I will put some link towards pages of Openclassroom that are nicely done, even though it's in French (good way to practise 😉).

The idea with this document is to have a grasp on the different concepts and word we might need during the TP, not to be a pro in Python.

SHORTCUTS AND SYNTAX

The first things to know are probably a few keys that will help you do things in a simple manner. And for that you need to know a bit of syntax.

COMMENT

Python will read and try to execute each line of your code. When your code is growing bigger, you might want to organize it so that you remember what the different parts of your code do. To organize your code, you can annotate it by writing comments. A commented line is a line that Python will not read: you can therefore write whatever you want.

To comment a line, you can either add the character "#" in front of the line or comment a group of lines.

```
# This line is commented
a = 10 + b     # This comment is inline
This line is normal code, and this is going to bug…
```

MULTI-LINE COMMENT

You can enclose a longer comment (like a description) in """. All the lines in-between will be commented.

```
"""
A description that can
Be splitted on many
Lines.
Ok."""
a = 10 + b
```

KEYBOARD SHORTCUTS

An IDE (Integrated Development Environment) is an environment that helps you write code. An IDE often offers syntaxic coloration (e.g. a comment is written in a different color), auto-completion, debugging mode, shortcuts, etc. Shortcuts help debugging faster, and as coding is mostly debugging, learning a few shortcuts will make coding more enjoyable…

DOCUMENTATION

A very nice feature of Python is that it's a very popular langage (it recently got the thrid place on the Tiobe index, beating C++…) and therefore Python benefits from a large and active community. Whenever you need to understand what a method does, many sites will help you (Python docs, w3schools, or even just google : simply write 'python' + the name of the function you're looking for).

Moreover, when you don't find the solution to a bug, it is possible somebody has had the same issue… Many forums (stackoverflow.com) are full of questions and people answering them. Again, all you need is to be able to formulate your question (which not always so easy) and give it to Google.

Shortcuts depend on the IDE so I suggest that when you start coding, you look up the Preferences and check how you can fast toggle comments/block comments, run a script, trigger auto-completion, get help on a method…

USEFUL FUNCTIONS

PRINT

To debug, you can use the debug mode of an IDE, which allows to set break points that, when reached, will pause the execution of your code. At that moment, you can check many things, including what's in your variables…

Another way to check is to display what's in your variable. To do that, you can use the function 'print'. (Of course, you can also use print to display stuff on the console during code execution, like 'Simulation has begun' to monitor which part of the code is being executed).

Fancy printing can be found here (not needed for the TP).

VARIABLES

Python is a very permissive langage : it is easy to get things done without really understanding what is happening, which makes degugging tricky. Example : variables. Without going into details (typically strings are not so important for us) this page is interesting to get the vocabulary and also to see the reserved words of Python (if you try to create a variable names 'True' you are going to have problems).

*It is important here to remember that '=' is used to attribute a value. What's left of '=' will receive the value of what's right of '='. The **boolean operator** that checks whether **two variables are equal** IS NOT '=' but '==' (see operators below).*

Variables

To declare a variable and assign a value to it, it's quite easy :

```
a = 1
```

I declared a variable named a which contains the integer 1.

FUNCTIONS AND METHODS

In Python, variables are objects. We will not go into details here, but this can clarify the distinction between functions and method. A method is applied to an object. A function does not require an object to be called.

An object is like a box that contains stuff and that has defined ways to manipulate it : methods. You call a method on an object to do something on it (for example, you can call the method 'append' on a list to add an element to that list, see lists). If my object is named 'listeA', this will be written : `listeA.append(myelement)`.

A function is not defined for a particular object and can be called directly, like 'len'. `Len(mylist)` is a function that returns the length of 'mylist', but this function is not called on an object.

Both functions and methods can have argument : an argument is indicated in parenthesis after the name of the function/method. For example, 'append' takes one parameter : the element to add to the list.

*To consult all the methods callable on an object, you can write 'nameoftheobject.' (don't forget the dot) and then press the key to trigger auto-completion (**CTRL+Space** on most IDEs).*

```
listeA.append(myelement)
```

Name of my **object**

Name of the **method** applied to my object 'listeA'

Name of the **parameter** given to the method 'append'.

Functions and method can also return a value. These values can be attributed to a variable in the same way you would attribute the value 15 to the variable 'a' for example :

```
a = 15      # assigns the value 15 to the variable a
b = [1, 2, 3]   # a list of 3 elements
print(a)    # will print 15
a = len(b)  # assigns the length of b to the variable a
print(a)    # will print 3
```

The documentation on a function/method gives you the parameters of a function and the returned values. There can be no argument and no returned value to a function.

LISTS

Lists are 1D vectors that help us store data. Manipulating lists often comes with bugs.

Tutorial : in the tutorial, tuples are also explained. They won't be useful to us, you can skip that.

*Index in a list : **the first element** of a list is at **index 0**. Mylist[2] returns the third element of mylist.*

Function and method	What it does
mylist.append('element')	**Appends 'element' at the end of a string**
mylist.insert(2, 'element')	**Insert 'element' at position 2**
Mylist.index(objectinmylist)	**Returns the index of objectinmylist in mylist**
Mylist.remove(objectinmylist)	**Removes objectinmylist from mylist**
Mylist.pop()	**Removes the last element of a list and returns it**
Len(mylist)	**Returns the length of mylist**
Range(x)	**Returns a list containing integers from 0 to x-1**

CONDITION

Operator	What it checks
<	Strictly inferior to
>	Strictly superior to
<=	Inferior or equal to
>=	Superior or equal to
==	Equal to
!=	Different from

Conditions can be useful to create events : for example IF simulation has run to half THEN add the second input to the reaction.

To do a condition, we can use a block called 'IF'.

More details on the 'IF' can be found here, with the different instructions (ELIF) and operators.

BLOCS ET INDENTATION

Au contraire de beaucoup (tous les autres ?) de langages, Python ne nécessite pas de balise fermante pour ses instructions en bloc et est donc **sensible à l'indentation**. Un bloc c'est par exemple une condition 'IF'. On signifie à Python qu'on va écrire une condition et le code qui suit la condition ne doit être réalisé que si la condition est vraie. Mais comment sait-il où s'arrête le code 'soumis à condition' ? Python n'a pas de balise de fin (par exemple END-IF) qui lui dit : 'Ok, après ça c'est plus sous condition'. Non, Python, lui, utilise l'indentation. After a block, you indent the code that is conditionned by the 'IF'.

```
IF condition #1:
   [Code executed if #1]
ELIF condition #2:
   [Code executed if #2]
[Code always executed]
```

All the code that is aligned to the IF/ELIF will **not** be conditioned and will always be executed.

A block instruction always has ':' at the end of it.

LOOPS

Loops are a way to repeat instructions. The main loop we are going to use is the FOR loop. The FOR loop is also a block. It will repeat the instructions in the block for a given number of time.

Loops

A common FOR loop consists in looping over the element of a 'range' list :

```
                # i will take the value 0,
for i in range(5):  # 1, 2, 3 and 4
                successively
                # will print 0, then 1,
then
      Print i   #  2, then 3, then 4
```

FILES

Manipulating files can come in handy when you have data that you want to exploit.

Files

```
# here, replace x with r (read), w(write, erases
# what was previously in the file) or a (append,
# you write at the end of the existing file)
```

In a nutshell :

```
myfile = open('pathToFile', 'x')



content = myfile.read()
myfile.write(stuff)
myfile.close()
```

USER DEFINED FUNCTIONS

Functions can help you avoid code duplicate. If there are a set of actions that you repeat many times, you might want to not rewrite the same lines of code each time. Functions also allow to split treatment in different scripts : a script with the functions that define how to run a simulation and a script where an actual system is described. With that, you can define different systems that use the same functions to run the simulation.

*Variables defined in a function are **local** and do not exist outside of it : you can't use `tmpVariable` outside of `myfunction` unless you declare it and put the returned value of `myfunction` into this newly defined variable.*

To that end, we need to define our own functions.

Functions

In a nutshell

```
def myfunction(argument1, argument2 = 15) :


""" Here, argument1 is mandatory when calling the function whereas
argument2 has a default value, which can be overwritten when calling
the function
"""       tmpVariable = argument1 + argument2
          return tmpVariable
                        # not every function returns variables



""" The following can be in another file (you need to import the
module with the functions)
"""
                                # myVariable is 27
                                # myVariable is 5
myVariable = myfunction(12)
myVariable = myfunction(2, 3)
```

LIBRARIES

Libraries contain functions. Librairies often regroup functions around a theme, like numpy consists in functions for scientific purposes. The Python community has come up with many libraries that can be imported in Python. Some IDEs come in with pre-downloaded libraries, otherwise you have to find the library you want online.

You can also define your own library that you can import in your projects.

Importing is quite simple :

```
Import numpy
```

All the functions imported this way can be called by adding the name of the library in front of the name of the function :

```
a = numpy.arange(5, 10)
```

you can alias the name of the library :

```
import numpy as np
a = np.arange(5, 10)
```

or even import them at the root, meaning that if the library contains a function that has the same name than a pre-existing function, the pre-existing function will be erased. By this way, the functions of the library can be called without the name of the library :

```
from numpy import *
a = arange(1, 5)
```