TP 1 : JupyterLab - Introductory Module

1 - Jupyter

Jupyter Lab is a web application for creating and running notebooks like this one. These notebooks are organized in the form of a list of cells. In the scope of this lecture we will use only cells of two types:

- *Markdown* cells (such as this block). It is written in *Markdown* format which allows to add short instruction in the text you want to display to define the layout;
- *Code* cells which can contain a computer program, or a part of a program. Each cell can be "ran, and include the display of some results. In this course, we will only use cells that allow the execution of python source code.

To summarize a bit roughly, each *code* cell is executed as if you were running a program containing the source code of all previously executed cells. Each cell actually shares the same memory and variable naming space.

Once you get used to it, a notebook is very useful because it allows to mix instructions, examples and programs that you can modify on the fly, and see the result by simply clicking on the "run" button.

I-1 Cells in jupyter notebook

A cell can be in several states.

- It is active by clicking once on it.
- By double-clicking on it, it becomes editable and you can modify its content.
- You run it by pressing the "run" button.

The active cell is the one next to which you see a colored vertical sidebar.

Exercise:

- Right click on the cell and look at the list of possible actions.
- Press "+" to create a new cell. Change its type (Markdown or code) by selecting it from the menu at the top of this notebook.
- Use the arrows on the top menu to move this cell.

Exercise :

This is a new markdown cell.

```
In [1]: "This is a new code cell"
Out[1]: 'This is a new code cell'
```

I-2 *Markdown*, a lightweight markup language

- All the tags that can be used in Jupyter are visible for example there (https://github.com/adam-p/markdown-here/wiki/Markdown-Cheatsheet).
- It becomes very powerful in the scope of this course because you have have the possibility to use the syntax of the $\LaTeX$ language for all mathematical equations and formulae.
- For instance, $\frac{dx}{dt} = -cx$ is written `$\frac{dx}{dt}=-cx$` (double click to edit this cell and check).
- Titles of different depths are simply obtained by adding one or more `#` depending on the desired depth, followed by a space.
- The display in *italic* (or **bold**) of a word or part of a sentence is obtained by adding `*` (or `**`) before and after it.
- List or numbered list are obtained using `-` or `1.` followed by space at the begining of the sentence.

Exercise : Create a *Markdown* cell which contains a text with terms in bold and italic, a list, 2 titles of different depth (If needed, edit cells above to check how things are done).

Exercise :

This is a first title

This is a second title

This setence is in bold. *This setence is in italic.*

- Here is a list.
- This is the second element in the list.
- And this is the last one.

---

2 - Python - The basics

We will use here the Python 3 language. It is an interpreted (as opposed to compiled) programming language. In fact it is not the "fastest" one, but it has a wide range of libraries that allow to do all the things we will need in this course. We limit ourselves here to an introduction to the basic instructions and principles, with the understanding that most of the exercises will only require modifications to the source code that we will provide.

2-1 Basic operation using *Code* cell

You can perform mathematical calculations, simply by using the operators `+` (addition), `-` (subtraction), `*` (multiplication), `/` (division) and `**` (power). If necessary and to avoid any problems with the order of operators. Be careful, if you put several calculations in the same cell, only the result of the last line is displayed. The reason for this is that the results are not explicitely printed and only the last one is stored by default. Of course, you must use the command print or use variables to store the result as we will see in the next paragraph.

Exercise: Execute the following cells, checking from the result that it is the one you expected, taking into account the priority rules and the parentheses.

```
In [2]: 1+1
Out[2]: 2

In [3]: 2*3
Out[3]: 6

In [4]: 3+4/2
Out[4]: 5.0

In [5]: 8-9
Out[5]: -1

In [6]: (1+2)*3+4
Out[6]: 13

In [7]: (1+2)*(3+4)
Out[7]: 21

In [8]: 1+2*3+4
Out[8]: 11
```

In [9]:
```
1+2*(3+4)
```
Out[9]: 15

In [10]:
```
(1+2*3+4)
```
Out[10]: 11

In [11]:
```
2**4
```
Out[11]: 16

In [12]:
```
2**(5-1)
```
Out[12]: 16

Exercise : I have executed all the code cells and results were the ones I expected.

2-2 The variables

- In order to do more complex operations than those seen above it is necessary to store the results.
- A variable is in a simplified way a label or a name that allows us to say on which value to do an operation and from which the interpreter is able to find it in memory.
- The name of a variable can contain any alphanumeric character, but it must not start with a number. Capital letters and accented characters are to be avoided by convention or at the risk of making errors that are sometimes difficult to identify. The name can in theory contain a very large number of characters, but this is obviously not recommended. The only special character allowed is the underscore `_` .
- The affection operator is the equal sign (not to be confused later with the equality operator in the mathematical sense). In the expression `x=10` , the value `10` to the right of the `=` operator is assigned in memory to the variable whose name `x` is to the left.
- An operation can then be done using the value of another variable. Thus `y=x+1` adds the value `1` to that of the variable `x` and then stores it in the variable `y` .
- The command `print(y)` displays the content of the variable `y` .

In [43]:
```
x=10
y=x+1
print("1: y=",y," x=",x)
```
```
1: y= 11  x= 10
```

- In the example above, you will have noticed that the commands are executed sequentially (i.e. one after the other) from top to bottom.
- Changing the value of a variable after it has been used does not change the values of the other variables even if they were calculated from it.
- It is the result of the operation with the values of the different variables at the time of the operation that is assigned to the variable storing the result.
- If you want to "update" the result (as a spreadsheet program does, for example), you must execute the corresponding command again, as in the example below.

Exercice: Execute it and check that you understand the different displays of the variables "x" and "y".

In [13]:
```
x=10
y=x+1
print("1: y=",y," x=",x)
x=37
print("2: y=",y," x=",x)
y=x+1
print("3: y=",y," x=",x)
```

```
1: y= 11  x= 10
2: y= 11  x= 37
3: y= 38  x= 37
```

Exercise : At first, x equals to ten and y equals to x plus one so it equals to ten. Then, the value ofthe variable x changes to thirty-seven. So, x equals to thirty-seven and y still equals to eleven because eleven is already stored in the variable y so the value of y does not depend of x anymore. Finally, the value x plus one is assigned to the variable y so y equals to thirty-eight.

Warning:

- The variables assigned during the previous executions of code cells are **known** from the start of the execution of a new cell, with **the last value** that was assigned to it.
- The order of execution of the cells is then of primary importance.
- This can lead to unexpected behavior (mispelling a variable name may not generate an error at runtime if the mispelled name is already known).
- To start from a "clean slate" and an execution history that is easier to interpret, it is sometimes useful to "restart the kernel".
- Concretely, doing this restarts a blank python execution environment whose memory is empty of any previous assignments.

Exercice:

- Execute the following cells

In [14]:
```
print(x,y)
```

```
37 38
```

In [15]:
```
z=2
y=y+x
print(y)
```

```
75
```

- restart the kernel, and execute the next cell.

In [1]:
```
x=1
z=2
y=y+x
print(y)
```

```
---------------------------------------------------------------------
-----------
NameError                                 Traceback (most recent
call last)
<ipython-input-1-3cef29ce12fc> in <module>
      1 x=1
      2 z=2
----> 3 y=y+x
      4 print(y)

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

- propose a correction so that the execution does not induce any more NameError by using the variable z judiciously

Exercise :

In [2]:
```
x=1
z=2
y=z+x    #We assign the value z plus x instead of y plus x becau
se the variable y was not assigned to a value after restarting t
he kernel
print(y)
```

```
3
```

2-4 Built-in Types

- So far we have mostly used variables to store integers. Obviously, this is a special case. Python has variables of different types to store different objects or variables.
- Python does not require that the type of a variable be imposed once and for all within an execution environment.
- It is possible to change the type of a variable at any time by reassigning a new value to it (derived or not from the previous value).
- the function `print(type(myvariable))` displays the type of the variable `myvariable` .
- for numbers we will use mostly integers (type `int`) and real numbers (type `float`).
- for some logical mathematical operations and manipulation of more complex structures like arrays, we will use boolean variables (type `bool`), whose possible values are `False` or `True` (with a capital letter).
- for strings, there is a string type (type `str`), and it is possible to manipulate a list of variables with the `list` type.

Exercise : Check in separate cells the type of each of the following : `100` , `3.14159` , `x` , `False` , `"COVID19"` , `"100"` , `"z=x+2"` , `[1, 2, 3]` , `(1, 2, 3)` .

Exercise :

In [3]:
```
var = 100
print(type(var))
```
```
<class 'int'>
```

In [4]:
```
var = 3.14159
print(type(var))
```
```
<class 'float'>
```

In [5]:
```
var = x
print(type(var))
```
```
<class 'int'>
```

In [6]:
```
var = False
print(type(var))
```
```
<class 'bool'>
```

In [7]:
```
var = "COVID19"
print(type(var))
```
```
<class 'str'>
```

In [8]:
```
var = "z=x+2"
print(type(var))
```
```
<class 'str'>
```

In [9]:
```
var = [1, 2, 3]
print(type(var))
```

```
<class 'list'>
```

In [10]:
```
var = (1, 2, 3)
print(type(var))
```

```
<class 'tuple'>
```

3 - Lists and Strings

3-1 Lists

A list is an ordered sequence of elements.

- `a=[3, 5, 7, 11, 16, 3.14159]` is a list named `a`, which contains the integers 3, 5, 7, 11,16 and the real 3.14159. They are separated by commas and enclosed in square brackets.
- This list contains 6 elements. We can get the number of elements in a list by using the command `print(len(a))`

In [12]:
```
a=[3,5,7,11,16,3.14159]
print(len(a))
```

```
6
```

- An element can be added to a preexisting list using this syntax

In [13]:
```
x=8
print(a)
a.append(12)
print(a)
a.append(x)
print(a)
```

```
[3, 5, 7, 11, 16, 3.14159]
[3, 5, 7, 11, 16, 3.14159, 12]
[3, 5, 7, 11, 16, 3.14159, 12, 8]
```

- you can access the elements of a list `a` using the index of each element between brackets. Indexes start at 0 (included) and end at `len(a)-1` (included)

In [14]:
```
print(a)
print("element of index=0 of list a",a[0])
print("element of index=5 of list a",a[5])
```

```
[3, 5, 7, 11, 16, 3.14159, 12, 8]
element of index=0 of list a 3
element of index=5 of list a 3.14159
```

- indexes start at 0 and end at len(a)-1
- it is possible to select several elements on a range of indexes, using the `:` separator.
- `a[imin:imax]` is a list of elements with index between `imin` (included) and `imax` (excluded) the following instructions are therefore equivalent.
- The values of `imin` and `imax` are optional

In [15]:
```
print(a)
print(a[0:len(a)])
print(a[:len(a)])
print(a[0:])
print(a[:])
```

```
[3, 5, 7, 11, 16, 3.14159, 12, 8]
[3, 5, 7, 11, 16, 3.14159, 12, 8]
[3, 5, 7, 11, 16, 3.14159, 12, 8]
[3, 5, 7, 11, 16, 3.14159, 12, 8]
[3, 5, 7, 11, 16, 3.14159, 12, 8]
```

- It is also possible to create a list from two pre-existing lists using the `+` operator

In [36]:
```
a=[1,9,13]
b=[5,7,3,2]
c=a+b
print(a,"+",b)
print("=",c)
```

```
[1, 9, 13] + [5, 7, 3, 2]
= [1, 9, 13, 5, 7, 3, 2]
```

Exercise:

- Create a list `name` containing the letters of your first name
- Create a list `prime` containing the first 5 prime numbers
- Display the first and last element of each of these lists with a command that will work for any list of length.
- Displat the second and third elements using the range syntax

Exercise :

In [22]:
```
name = ["m", "y", "r", "i", "a", "m"]
prime = [2, 3, 5, 7, 11]
print("The first and the last elements of each list are :", name
[0], ",", name[len(name)-1], "and", prime[0], ",", prime[len(pri
me)-1],".")
print("The second and the third elements of each list are :", na
me[1], ",", name[2], "and", prime[1], ",", prime[2],".")
```

```
The first and the last elements of each list are : m , m and 2 ,
11 .
The second and the third elements of each list are : y , r and 3
, 5 .
```

3-2 (characters) Strings

We can indicate a character string by using either quotes (for example, "James") or apostrophes ('James'). Strings are actually a type of immutable list. It is therefore possible to access a particular character from the list by its index.

Exercise: Create the string `s` containing your name. Display the second element of the list (with index 1) and try to modify this last by another letter. What do you get ?

Exercise :

In [23]:
```
s = "MABROUKI"
print(s[1])
```

```
A
```

In [24]:
```
s[1] = "Z"
```

```
---------------------------------------------------------------
-----------
TypeError                                 Traceback (most recent
call last)
<ipython-input-24-844a8e4649fa> in <module>
----> 1 s[1] = "Z"

TypeError: 'str' object does not support item assignment
```

It is written that str object does not support item assignment, so it means strings are immutable.

Exercise: Using the + operator, concatenate the strings of your first name and that of your last name preceded by a blank. Multiply (*) the character string of your first name by 6

In [33]:
```
first_name = "Myriam"
last_name = "MABROUKI"
name = first_name + " " + last_name
print(name)
first_name * 6
```

```
Myriam MABROUKI
```

Out[33]:
```
'MyriamMyriamMyriamMyriamMyriamMyriam'
```