



# Python

*(a crash course)*

Pedro Barahona

2019 / 20

# Introduction

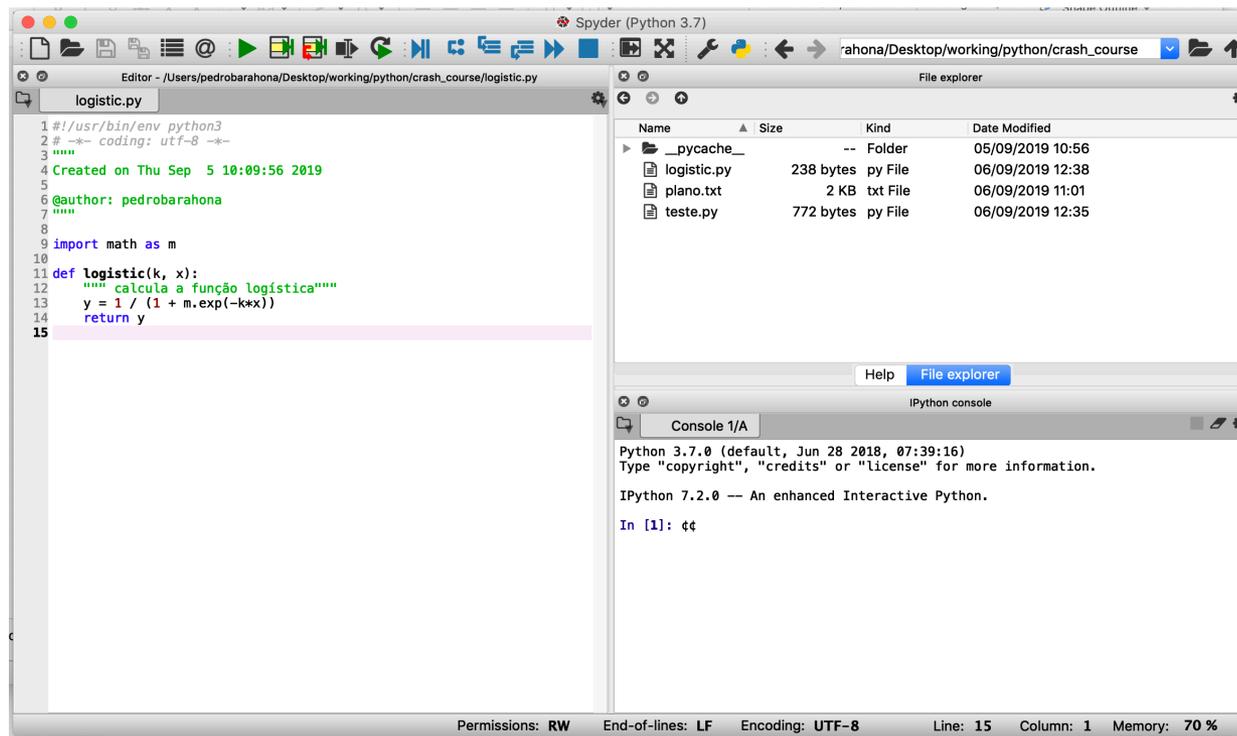


- This introductory (crash) course to Python is aimed at students of master programmes that require some programming skills in this language, and accept students with various backgrounds.
- This “hands on” course uses the Anaconda software, namely its Spyder IDE, that should be downloaded and installed by the students from
  - <https://www.anaconda.com/distribution/>
- More information can be obtained from several text books, namely
- Allen B. Downey. Think Python: How to Think Like a Computer Scientist.
  - <http://greenteapress.com/wp/think-python-2e/>
- John V. Guttag. Introduction to Computation and Programming Using Python, MIT PRESS, 2016.
  - <https://mitpress.mit.edu/books/introduction-computation-and-programming-using-python-second-edition>

# Spyder



- The Spyder IDE should be configured to show 3 windows: Editor, File Explorer and Console



# Spyder



- The Console may be used as a simple calculator, using Python instructions.

A screenshot of the IPython console window within the Spyder IDE. The window title is "IPython console" and it has a tab labeled "Console 1/A". The console output shows the Python version (3.7.0), the IPython version (7.2.0), and a series of interactive commands and outputs. The first command is `x = 2*3`, the second is `x` which outputs `6`, and the third is `|` indicating the prompt is ready for input. The status bar at the bottom shows "End-of-lines: LF", "Encoding: UTF-8", "Line: 15", "Column: 1", and "Memory: 70 %".

```
Python 3.7.0 (default, Jun 28 2018, 07:39:16)
Type "copyright", "credits" or "license" for more information.

IPython 7.2.0 -- An enhanced Interactive Python.

In [1]: x = 2*3

In [2]: x
Out[2]: 6

In [3]: |
```

End-of-lines: LF    Encoding: UTF-8    Line: 15    Column: 1    Memory: 70 %

# Python – Basic Types



- Python implements several basic types namely
  - int e.g. 1578
  - float e.g. 24.57
  - bool e.g. True
  - char e.g. "c"
- Elements with these types can be used in more complex data structures, such as lists and strings.

# Python – Assignment



- A programme is a sequence of instructions. The most basic instruction is the assignment

`var = expression`

- That assigns to variable `var`, the value of the expression.
- Expressions can be obtained from built-in operators with the usual precedence and parentheses

`x = (2 + y) * 4`

- Expressions may also use functions, either built-in or user defined,

`x = xpto(a, 5)`

- Many functions are available in specialised libraries. Possibly the most used library is the math library that implements a number of useful mathematical functions.
- Libraries must be imported prior to their use, as shown

```
In : import math as m
```

```
In : x = m.sqrt(25)
```

```
Out: 5.0
```

- The functions supported by the libraries can be obtained with the command `dir`.

```
In : import math as m
In : dir(m)
Out: ['__doc__',
      '__file__',
      '...',
      '__spec__',
      'acos',
      'acosh',
      'asin',
      '...',
      'trunc']
```

- Other useful libraries are:
  - NumPy – Supports N-dimensional arrays, and functions such as Iteration, Fourier Transform, linear algebra and financial functions.
  - SciPy – supports scientific calculations such as integration, optimization and sparse eigenvalues.
- More on useful libraries in
  - [https://linuxhint.com/10\\_best\\_math\\_libraries\\_python/](https://linuxhint.com/10_best_math_libraries_python/)

- Functions in Python are defined with the following syntax:

```
def FunctionName(parameters):  
    """ Function documentation """  
      
    Instruction block  
    return [None | result(s)]
```

Although optional, the **documentation** is very useful to elucidate the purpose of the function and is available with the command

- `help(functionName)`

# Python – Functions



```
def FunctionName(parameters):  
    """ function documentation  
    """  
  
    instruction block  
  
    return [None | result(s)]
```

- The parameters are input for the function and are separated by commas.
- (If there are none, the function is a *constant*.)
- The instruction block is a set of instructions, adequately **indented** and **commented**, that manipulate the input parameters and internal variables and possibly yield some result(s).

# Python – Functions



```
def FunctionName(parameters):  
    """ function documentation  
    """  
  
    instruction block  
  
    return [None | result(s)]
```

- The value of the function is specified with the return instruction.
- A function might return no results and be used only to produce some side effect.
- For example function **print()** does not return any value, but only shows the results in the console.

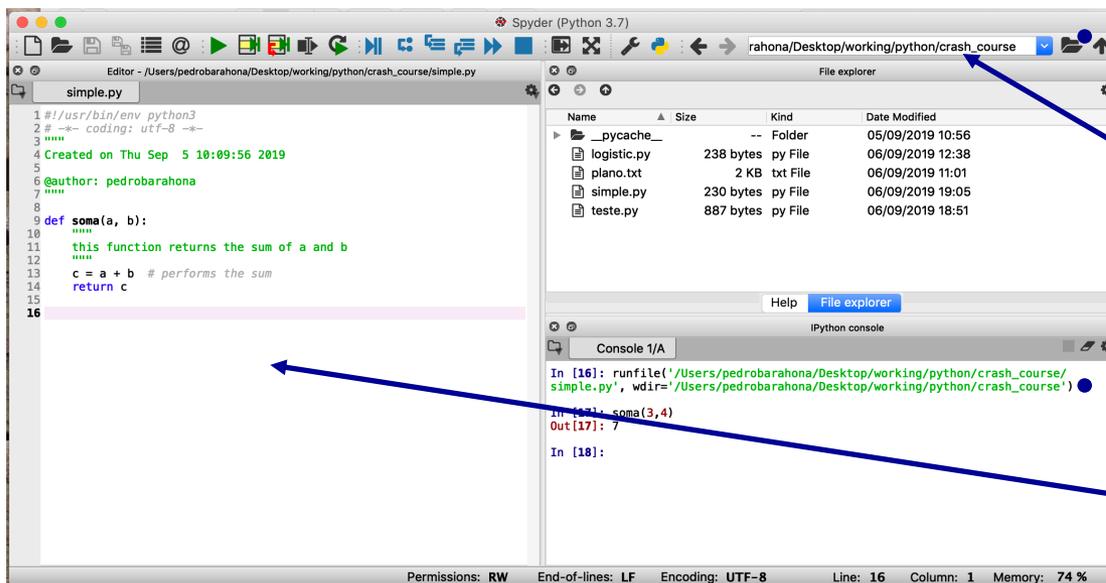
- Example:

```
def soma(a, b):  
    """  
    this function returns the sum of a and b  
    """  
    c = a + b # performs the sum  
    return c
```

- The function introduces a local variable `c`, which is not seen outside the function.
- The indentation of the instruction block is **mandatory**.

# Functions - Spyder

- Functions should be written in a text file with extension `.py`, and become visible by making their directory as the **current directory**.

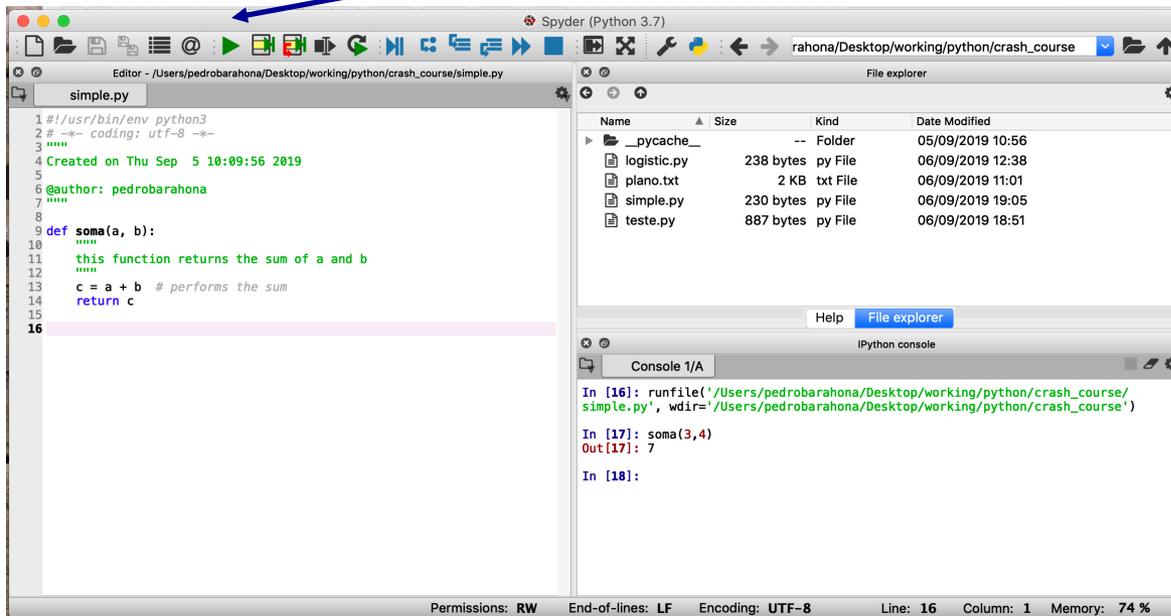


Selecting the current directory is done in the *File Explorer* window of Spyder.

Files can be edited in the *Editor* window of Spyder.

# Functions - Spyder

- Once they are edited and in the current directory, function should be loaded so that they can be called from the console.
- Loading can be done with the command `runfile(filename)` or simply by clicking on the run icon. 



```

1 #!/usr/bin/env python3
2 #-*- coding: utf-8 -*-
3 """
4 Created on Thu Sep 5 10:09:56 2019
5
6 @author: pedrobarahona
7 """
8
9 def soma(a, b):
10
11     this function returns the sum of a and b
12     '''
13     c = a + b # performs the sum
14     return c
15
16

```

```

In [16]: runfile('/Users/pedrobarahona/Desktop/working/python/crash_course/
simple.py', wdir='/Users/pedrobarahona/Desktop/working/python/crash_course')

In [17]: soma(3,4)
Out[17]: 7

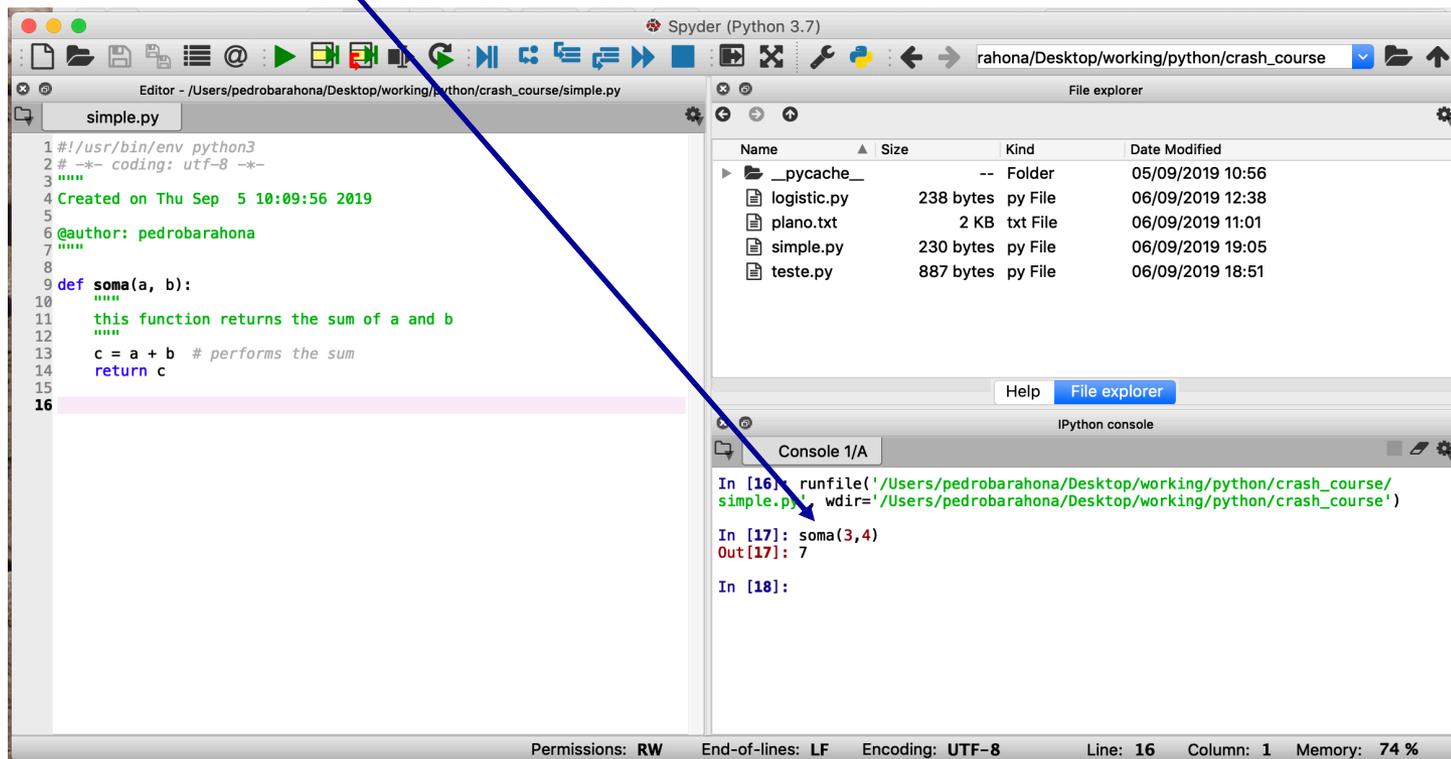
In [18]:

```

# Functions - Spyder



- Once loaded a function can be called from the console, as shown below.



- **Exercise 1a:** Write a functions that implement the logistics (sigmoid) function, with parameters  $L$ ,  $k$  and  $x_0$ , for any real value  $x$ .

## Logistic function

From Wikipedia, the free encyclopedia

A **logistic function** or **logistic curve** is a common "S" shape ([sigmoid curve](#)), with equation:

$$f(x) = \frac{L}{1 + e^{-k(x-x_0)}}$$

where

- $e$  = the [natural logarithm](#) base (also known as [Euler's number](#)),
- $x_0$  = the  $x$ -value of the sigmoid's midpoint,
- $L$  = the curve's maximum value, and
- $k$  = the logistic growth rate or steepness of the curve.<sup>[1]</sup>

- **Exercise 1b:** Write a functions that implement the Gauss (bell shaped) function with parameters  $\mu$  and  $\sigma$  for any real  $x$ .

## Gaussian function

From Wikipedia, the free encyclopedia

Gaussian functions are often used to represent the [probability density function](#) of a [normally distributed random variable](#) with [expected value](#)  $\mu = b$  and [variance](#)  $\sigma^2 = c^2$ . In this case, the Gaussian is of the form:

$$g(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2}.$$

Gaussian functions are widely used in [statistics](#) to describe the [normal distributions](#), in [signal processing](#) to define [Gaussian filters](#), in [image processing](#) where two-dimensional Gaussians are used for [Gaussian blurs](#), and in mathematics to solve [heat equations](#) and [diffusion equations](#) and to define the [Weierstrass transform](#).

# Python – Lists



- Python provides the data structure **list**, to allow the organization of collection of any type of objects, not only of simple data types (e.g. Int or float) but also other more complex objects, such as lists.
- Instances (objects) of this type of data structure (class) are typically created with simple **enumeration**. For example,
  - `L = [1,2,3,4]`
  - `M = [1, "a", [1,2,3]]`
  - `S = ["a", "b", "c"]`
- The last case, a list of characters is usually created as a string,
  - `S = "abcd"`

# Python – Lists



- Before using a list, it is *convenient* to initialise it, which can be done with the repetition instruction.

```
In : L = [0]* 5
In : L
Out: [0,0,0,0,0]
In : [None]*3
In : [None, None, None]
```

- Lists are “mutable” objects, in that they can be **appended** with extra elements, **extended** with other lists, or have elements removed.
- Methods for list objects are available to perform these changes.

# Python – Lists



- In general, existing methods available for an object may be consulted with the **dir** command.

```
In : L = [0]* 5
In : dir(L)
Out:
['__add__',
 .....
 '__len__',
 .....
 'append',
 'copy',
 'extend',
 .....
 'remove',
 'reverse',
 'sort']
```

- Some examples:

```
In : L = [1,2,3,4]
In : M = [6,8,7,8]
In : L.append(5)
In : L
Out: L = [1,2,3,4,5]
In : L.extend(M)
In : L
Out: L = [1,2,3,4,5,6,8,7,8]
In : L.remove(8)
In : L
Out: L = [1,2,3,4,5,6,7,8]
```

# Python – Lists



- Lists are not sets, in that elements of the list have a position (index).
- Indices in a list of length  $n$ , range from 0 to  $n-1$ . Elements of a list can be accessed by means of their index, either positive (0 to  $n-1$ , from left to right) or negative (from  $-1$  to  $-n$ ) from right to left.
- The length of a list can be obtained with method **len**.

```
In : L = [1,2,3,4]
In : len(L)
Out: 4
In : L.__len__()
Out: 4
In : L[2]
Out: 3
In : x = L[-3]
In : x
Out: 2
```

# Python – Lists



- Quite often, it is convenient to obtain not a single element but a **slice** of the list, specified with notation

[start : end : step]

- The slice is composed of all elements starting with that of index start, up to (and excluding) that with index end, spaced by an optional step (default is 1)

```
In : L = [1,2,3,4,5,6]
In : L[2:5]
Out: [3,4,5]
In : L[0:5:2]
Out: [1,3,5]
In : L[-1,3,-1]
Out: [6,5]
```

# Python – Lists



- Lists are mutable objects in that their state may change.
- Not only the lists can be extended and “shrunked” as seen before, but also their elements may change.

```
In : L = [1,2,3,4,5,6]
In : L[3] = 9
In : L
Out: [1,2,3,9,5,6]
```

# Python – Tuples



- Tuples are similar to lists. They can be created by enumeration with brackets notation.
- Tuples are immutable objects. Once created they can not be changed.

```
In : T = (1,2,3,4,5,6)
In : T[1]
Out: 2
Out: T[1] = 9
TypeError: 'tuple' object does not support item assignment
```

- Methods available to tuple objects can be obtained with the command **dir**.

# Python – Sets



- Sets are also similar to lists, but
  - a. their elements are not accessible by indices.
  - b. they do not take repeated elements.

```
In : S = {1,2,3,1}
In : S
Out: {1,2,3}
Out: S[1]
TypeError: 'set' object does not support indexing
```

- Methods available to set objects can be obtained with the command **dir**.
- Sets are useful to implement **dictionaries** (later).

- Matrices (and higher order arrays) can be implemented as lists of lists. Their elements can be reached as before.

```
In : M = [[1,2,3,4],[4,5,6,7]]
In : len(M) # number of rows
Out: 2
In : len(M[0]) # number of columns
Out: 4
In : M[1][2]
Out: 6
```

- Although all matrix operations can be implemented with nested lists, library NumPy is very useful for linear algebra operations on vectors and arrays (later).

# Python – Conditional Execution (IF)



- As in all imperative languages a block of instructions may be executed conditional, i.e. if a certain condition is met. The syntax in Python of the IF instruction is

```
if condition_1:
    instructions block
elif condition_2:
    instructions block
...
elif condition_n:
    instructions block
else
    instructions block
```

- Note: Both the **elif** and the **else** parts of the instruction are optional.

# Python IF - Conditions

- A condition in Python is simply a Boolean expression, i.e. that evaluates to True or False.
- Simple conditions are usually obtained with the relational operators applied to (numerical) expressions

" == ", " != ", " > ", " >= ", " < ", " <= "

```
In : a = -3
In : if a > 0:
    ..:     c = a
    ..: else:
    ..:     c = -a
In : c
Out: 3
```

# Python IF - Conditions



- More complex conditions may be formed with Boolean expressions obtained with Boolean operators, conveniently parenthesised

" and ", "or ", " not "

```
In : L = [1,2,3,4]
In : if len(L) > 6 and L[6] > 0
    ..:     c = L[6]
    ..: else:
    ..:     c = -1
In : c
Out: 3
```

# IF – Roots of 2<sup>nd</sup> degree Equation



```
def roots_2(a,b,c):
    """ The function return the roots of the equation ax**2 + bx + c = 0
    """
    d = b**2 - 4 * a * c
    if d < 0:
        return []
    elif d == 0:
        return [-b / (2*a)]
    else:
        return [-b + m.sqrt(d)/ (2*a), \ # line continuation
                -b - m.sqrt(d)/ (2*a)]
```

# IF – Roots of 2<sup>nd</sup> degree Equation



- The function just defined should be tested, **for all possible conditions**, before it is used elsewhere

```
In : roots_2(1, 0, 4)
Out: []
In : roots_2(1, 0, -4)
Out: [2.0, -2.0]
In : roots_2(1, 1, -6)
Out: [-2.0, 3.0]
```

- **Note: UNITARY TESTS**, as above, should be done in all functions that are defined by the user, before being used in more complex programs.

## Exercise 2:

- Specify a function that checks the type of triangle is obtained with 3 segments of given lengths a,b, and c.
- The signature of the function should be

**def triangle(a,b,c):**

- The value that the function returns should be:
  - 3: equilateral triangle
  - 2: isosceles triangle
  - 1: scalene triangle
  - 0: not a triangle

# Python – Repeated Execution (FOR)



- In virtually all (non trivial) programs a block of instructions must be executed repeatedly, usually with some minor changes in each execution.
- The FOR instruction achieves this behaviour. Its syntax is

```
for var in sequence:  
    instructions block
```

- Every execution of the FOR block of instructions is executed with a different value of variable var.

# Python – Repeated Execution (FOR)



```
for var in sequence:  
    instructions block
```

- Sequences can be lists, or tuples or sets as defined before.
- They may also be **ranges** with syntax similar to that used in slices of lists

`range([start ,end ,step])`

- where
- 2 arguments: start and end. By default, step = 1.
- 1 argument: end. By default, start = 0 and step = 1.

# Python – Example FOR



**Example:** Obtain the sum of the elements of a list

- The first formulation iterates on the **elements** of the list.

```
def sum_list_1(L):  
    """ The function returns the sum of the elements of list L  
    """  
    s = 0  
    for v in L:  
        s = s + v  
    return s
```

- If the list has no elements, no iterations are performed and the function returns 0.

# Python – Example FOR



**Example:** Obtain the sum of the elements of a list

- The second formulation iterates on the **indices** of the list.

```
def sum_list_2(L):  
    """ The function returns the sum of the elements of list L  
    """  
    s = 0  
    for i in range(len(L)):  
        s = s + L[i]  
    return s
```

- Note the use of the length of the list. As before, if the list is empty (with length 0) the function returns 0.

# Python – Lists by comprehension



- The concept of iteration can also be used to create a list **by comprehension**, as shown in the examples.

```
In : L = [ a**2 for a in range(4)]
In : L
Out: [0, 1, 4, 9]
Out: M = [m.sqrt(i) for i in [0,1,4,9,16]]
In : M
Out: [0.0, 1.0, 2.0, 3.0, 4.0]
In : list(range(5))
Out: [0,1,2,3,4]
```

## Exercise 3a:

- Specify a function that returns the numbers of the Fibonacci series:  
1, 1, 2, 3, 5, 8, ...

- The signature of the function should be

**def fibo(n):**

- The function returns the  $n^{\text{th}}$  number of the series.
- For example  $\text{fibo}(6) = 8$

## Exercise 3b:

- Specify a function that obtains the sum of the powers **p** of the first  $n$  integers (starting at 1).
- The signature of the function should be

**def sum\_powers(n, p):**

- The function returns the sum of the p-powers of the  $n$  first natural numbers.

$$1^p + 2^p + \dots + n^p$$

## Exercise 3c:

- Specify a function that returns the scalar product of 2 arrays (assume they have the same size).
- The signature of the function should be

**def scalar\_prd(A, B):**

- For example, if the function is called with arrays

$M = [1,2,3,4]$  and  $N = [8,7,6,5]$

it should return the value

$$1*8+2*7+3*6+4*5 = 60$$

## Exercise 3d:

- Specify a function that returns the transpose of a matrix implemented as a list of lists.
- The signature of the function should be

**def transpose(M):**

- For example, if the function is called with matrix

[ [1,2,3,4] , [5,6,7,8] ]

it should return matrix

[ [1,5] , [2,6] , [3,7] , [4,8] ]

## Exercise 3e:

- Specify a function that returns the product of 2 matrices implemented as lists of lists (with compatible dimensions).
- The signature of the function should be

```
def mat_prod(M, N):
```

```
    [ [1,2,3,4] , [5,6,7,8] ]
```

- For example, if the function is called with matrices

```
M = [ [1,2,3] , [4,5,6] ] and N = [ [1,2,3,4] , [9,8,7,6] , [0,1,2,3] ]
```

it should return matrix

```
[[19, 21, 23, 25],[49, 54, 59, 64]]
```

# NumPy – Array Library



- As mentioned earlier linear algebra is supported by library NumPy. The library should be imported (usually as np) and its classes consulted with the dir() command. Examples:

```
In: import numpy as np
In: M = np.array([[1,2,3],[4,5,6]])
In: np.transpose(M)
Out:
array([[1, 4],
       [2, 5],
       [3, 6]])
In: N = np.array([[1,2,3,4],[9,8,7,6] [0,1,2,3]])
In: Q = np.matmul(M,N)
Out:
array([[19, 21, 23, 25],
       [49, 54, 59, 64]])
```

# Python – Repeated Execution (WHILE)

---

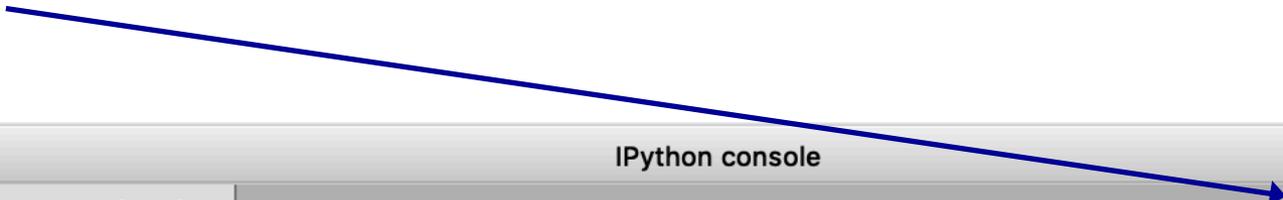
- In some cases, it is not known how many times the block of instructions should be iterated. Only that it should do so **while** a certain condition is verified.
- The WHILE instruction achieves this behaviour. Its syntax is

```
while condition:  
    instructions block
```

- Of course, the instructions in the block should change the state of the condition, eventually making it False.
- If the condition is False before the instruction, then no instance of the block is executed.

# Python – Repeated Execution (WHILE)

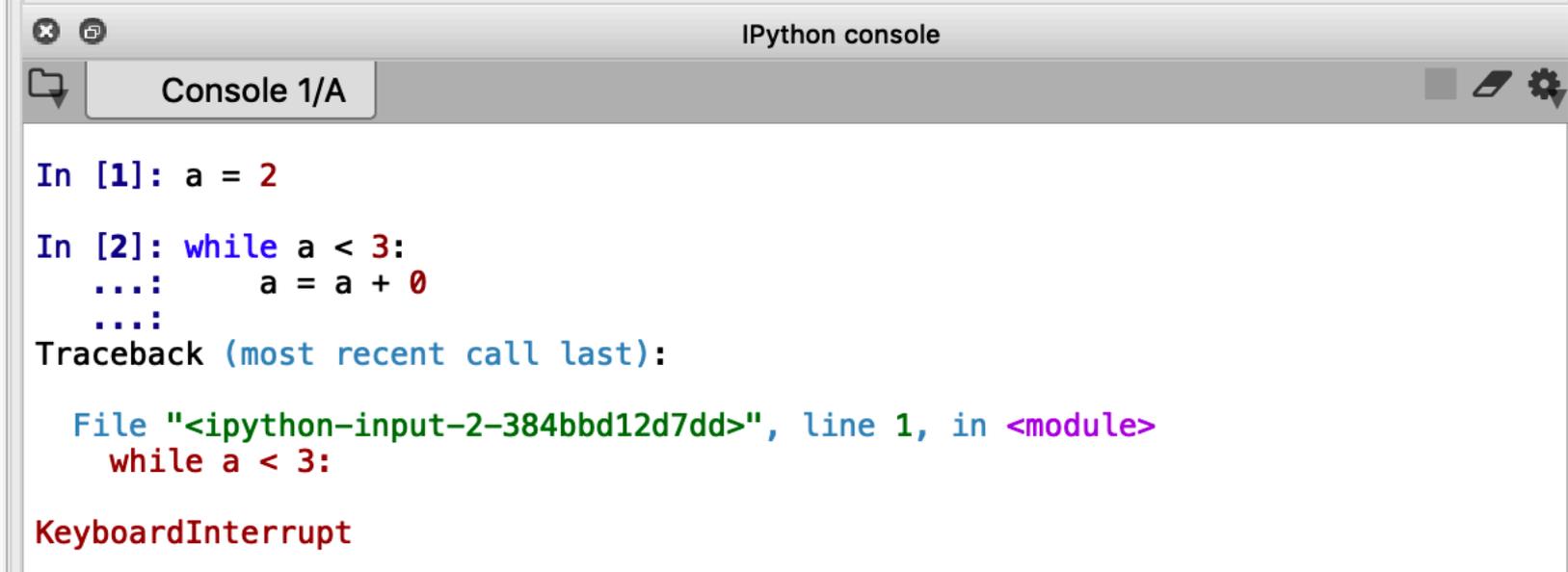
- Note: Care must be taken so that the condition eventually become false, otherwise the program enters an infinite loop.
- In this case, execution may be aborted by clicking on the suspend icon



```
IPython console
Console 1/A
In [1]: a = 2
In [2]: while a < 3:
...:     a = a + 0
...:
```

# Python – Repeated Execution (WHILE)

- Note: Care must be taken so that the condition eventually become false, otherwise the program enters an infinite loop.
- In this case, execution may be aborted by clicking on the suspend icon



```
IPython console
Console 1/A

In [1]: a = 2

In [2]: while a < 3:
...:     a = a + 0
...:
Traceback (most recent call last):

  File "<ipython-input-2-384bbd12d7dd>", line 1, in <module>
    while a < 3:
KeyboardInterrupt
```

# Python – Repeated Execution (WHILE)

---

- Example: To find an element in a list, computation should stop as soon as the element is found.

```
def find_in_list(x,L):
    """ The function returns the index of the element x, if it belongs to
    the list. Otherwise returns -1
    """
    i = 0
    while i < len(L) and L[i] != x:
        i = i + 1
    if i >= len(L):
        return -1
    else:
        return i
```

# Python – Repeated Execution (WHILE)

---

- In fact, the condition signals an interrupt to the cycle. This interruption may often be achieved by using the return instruction when the condition is met.
- For example:

```
def find_in_list_2(x,L):  
    """ The function returns the index of the element x, if it belongs to  
    the list. Otherwise returns -1  
    """  
    for i in range(len(L)):  
        if L[i] == x:  
            return i  
    return -1
```

## Exercise 4a:

- Specify a function that returns the number of items (representing weights) of a list that are necessary to reach a certain weight  $w$ . It returns a pair indicating the number of items used (0 if  $w$  is not achieved), and the weight of the selected items.
- The signature of the function should be

**def select\_items(L, w):**

- Let  $L = [38, 25, 19, 11, 9]$ . Now if the function is called
  - with  $w = 80$  it should return the pair (3, 82).
  - with  $w = 200$  it should return the pair (0, 102).

## Exercise 4b:

- Specify a function that returns the sum of the series

$$1, 1/2^2, 1/3^2, 1/4^2, 1/5^2, \dots$$

with a *precision*  $p$  (i.e. the first neglected element is less than  $p$ ).

The signature of the function should be

```
def sum_inv_squares(p):
```

- Specify a similar function that returns the sum of the series

$$1, -1/2, +1/3, -1/4, +1/5, \dots$$

with signature

```
def alternated_harmonic(p):
```