Brought to you by:

ASTRA

PYTHON PRACTICE

1°YEAR BIEM / BIEF / BIG / BEMACC

Written by
Alessia Massara
Valentina Benzi
Curated by
Michele Longo

2022-2023 Edition

Find more at:

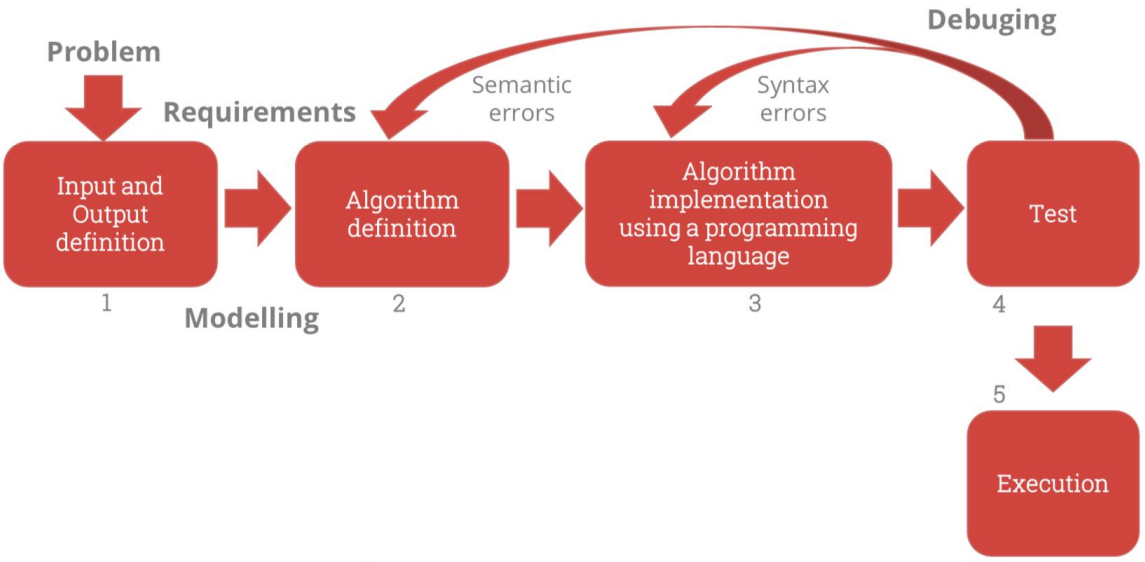
astrabocconi.it

Python

What is Programming?

Programming means instructing a machine how to perform a certain task or solve a certain problem.

When a program is developed, its realization cannot be improvised, in fact it requires several phases, starting from the analysis of the requests that the program must satisfy up to its writing.
To actually be able to provide the necessary instructions, the programmer must possess **technical skills**, i.e. he must know the logic of programming and of the specific language, and **not technical ones**, such as problem solving and computational thinking.



⇒ The **requirements** indicate the objectives to be achieved and the functionalities that the program will have to offer. →Fundamental requirements analysis.
⇒ In the **design phase** (definition of the Algorithm) the guidelines of the software structure are defined according to the requirements highlighted in the analysis. → Algorithms are used to represent the flow of instructions that the program will have to carry out to solve the problem.
⇒**Implementation** is the actual realization phase of the program: the software solution is realized through programming.
⇒ The software is tested so that any errors can be resolved through **debugging**.
Finally the program is executed and used.

What is an algorithm?

The **algorithm** is a process that allows us to solve a problem through a sequence of codified actions.

<u>Key features of the algorithms:</u>
- finite sequence of instructions;
- procedures must lead to a result;
- physically executable instructions;
- instructions expressed unambiguously.

Algorithms are commonly represented using **flowcharts**, a graphical notation to intuitively describe the actions of which the algorithm is composed.
- The **oval boxes** indicate the start and finish blocks;
- **Rhomboids** are input/output elements;
- **Rectangles** are execution elements = represent elementary actions;
- The **rhombuses** are elements of decision or choice;
- The **arrows** represent the execution flow of the algorithm.

What's Python: It's an **interpreted**, **interactive**, **object-oriented** high-level programming language.
- **Interpreted**
- **Interactive**: allows you to write instructions directly on its command prompt, the shell, without needing to create or modify a source file.
- **Object oriented:** the solution to a problem is not seen only as a sequence of instructions, but in terms of objects, their attributes and the actions that these objects can perform.
Other key features of Python are: clarity in its syntax (blocks of statements are delimited only by **indentation**) and dynamic typing of variables (since it is sufficient to assign an initial value to a variable to define its type).

IDLE = Integrated Development and Learning Environment → Python programming environment that includes:
- *the shell*: it is used in interactive mode, i.e. it allows you to type and execute single instructions and immediately see the result;
- *the editor*: it is used in script mode, i.e. it allows you to create actual programs that can be saved and executed.
The **shell** is the Python **interpreter**: the program that can read the code instructions and execute them. In case wrong instructions are entered, the interpreter will show an *error message*. ⇒ In the shell it is therefore possible to do all the desired experiments.
⇒ In the shell it is not possible to cancel what has already been done: if you want to start over from a clean page you will need to close and reopen the shell (*or "Restart Shell"*).
⇒ Statements entered in the shell are not stored, they simply produce a result.

The **editor**, allowing you to save the instructions, is the most suitable for developing programs. -> Running the program will start the interpreter in script mode: the code will be read and executed step by step until the program ends.

Python syntax basics:
The *syntax* of a program is that set of rules that must necessarily be known in order to write a program.

- Python is not strict on **space** management. →3+3 equals 3 + 3.
[it is preferable to use spaces to improve readability, although they are not mandatory]
- Each program has a set of **keywords**, each of which has a specific meaning and cannot be used for other purposes.

```
Here is a list of the Python keywords.  Enter any keyword to get more help.

False               class               from                or
None                continue            global              pass
True                def                 if                  raise
and                 del                 import              return
as                  elif                in                  try
assert              else                is                  while
async               except              lambda              with
await               finally             nonlocal            yield
break               for                 not
```

- In Python, you need to enclose **strings** in single quotes (‘ ‘) or double quotes (“ “).
  - If the string contains quotes then it must be enclosed in quotes and vice versa.
  - Three quotes or three quotes can be used to enclose strings that span multiple lines. ⇒ Often used for a program *presentation string*.
  - It is NOT possible to perform calculations using strings, even if their content is numeric.
  - **Operators usable with strings:** “ **+** “ for concatenation and “ ***** “ for repetition.

Comments can be useful as the program becomes more complex, to document the various steps. ⇒ They are short notes, which can be inserted in different points of the program and explain what objectives the instructions to which they refer have. → You can insert them in 2 ways:
1. When they follow a line of code, by preceding them with the symbol **#**;
2. When they are between different lines of code, in addition to the symbol **#**, also by inserting them between **triple quotes** (**or quotation marks**).

Escape codes are special commands consisting of a character preceded by a backslash (****), placed in a string. ⇒ These codes are not displayed in the output, but give specific commands:
- **\n** → wrap the text;
- **\t** → align text to tab stop;
- **** → wraps the code.

Python contains several built-in functions: **the built-in functions** (displayed in **purple**).
The most common *built-in* function is print = allows you to display an output on the screen.
[**ATTENTION!** The print function (but in general all functions) are written in **lowercase**, if they were written with uppercase (eg: Print or PRINT) an *error message* would come up.]

```
>>> print(
      print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)
```

Function parameters are always separated by a **comma** (parameter separator).
The first parameter of the **print** function is *value*, which is the value that will be returned as output.
→ One or more values can be entered separated by a comma.
Another parameter is **sep= ' '**: it indicates which separator must be returned between the various output texts. ⇒ The default separator is space.

The **help** function displays information about a function, data type, or module. → It can be used whenever you have doubts about one of these elements.

In addition to the pre-existing functions, Python also provides **libraries** that make it possible to expand its potential. ⇒ To access the functions of a specific library it is necessary to import it:

```
>>> import math
>>> math.pow(2,2)
4.0
```

Mathematical operators:
[ATTENTION! In Python, numbers follow the Anglo-Saxon notation: "**.**" for decimals and "**,**" for thousands.**]**

Symbol	Operation	Description
+	Addition	Adds two numbers
-	Subtraction	Subtracts two numbers
*	Multiplication	Multiplies two numbers
/	Division	Divides two numbers and returns a floating-point number
//	Floor division (aka integer division)	Divides two numbers and rounds down to an integer (*)
%	Modulus	Divides two integers and returns the remainder of the division
**	Exponentiation	Elevates a number to a power

A mathematical expression is a sequence of numbers, mathematical operators and parentheses that performs a calculation and returns a value (**DISCLAIMER!** It can contain both *numbers* and *variables representing numbers*).
In mathematical expressions in Python, some rules concerning the precedence of operators and parentheses are respected:
- operations enclosed in parentheses are performed first,
- Subsequently, when two operators share the same operand, the operator with the highest precedence is applied first. → Order of operators: exponentiation, multiplication and division, addition and subtraction.
[While in mathematical notation it is not always necessary to explain operators, in Python it is always necessary to explain operators, as well as parentheses.]

You are not always satisfied with the appearance of numbers displayed on the screen, especially in the case of **floating point numbers**. ⇒ you can format them with the **format** function:
The *format* function has two arguments:

```
>>> format(
        (value, format_spec='', /)
Return value.__format__(format_spec)
```

```
x=5.9/2.4
print(format(x,'.2f'))
```
→
```
2.46
>>>
```

- the number to be formatted;
- the **format specifier:**
  - ➔ '.2f' = to round to the second decimal number
  - ➔ ',d' = to insert the thousands separator;
  - ➔ '%' = to format a floating point number as a percentage (you can enter the number of decimal numbers you want: '.0%'

In Python there are several **built-in functions dedicated to mathematical calculations:**
- **sum** function: calculates the sum of a list of elements indicated in square or round brackets separated by a comma;
- **pow** function: calculates the power of a number;

```
>>> pow(
    (base, exp, mod=None)
    Equivalent to base**exp with 2 arguments or base**exp % mod with 3 arguments
```

- **abs** function: calculates the absolute value of a number;
- **round** function: rounds a floating point number to the decimal digit indicated as second parameter;
- **max** and **min** function: they return the maximum and minimum value of a series of single parameters.

In addition to the built-in functions, there are other **functions that are part of the Python standard library.** ⇒ Some of the additional modules are: **statistics** (in particular *statistics.mean* to calculate the mean of a list of values), **math** (ex: *math.sqrt* for the square root) and **random**, which allows to generate random numbers (ex: *random.random()* = decimal random number between 0 and 1, *random.randint(a,b)* = integer random number between (a and b included).

The variables

A **variable** is a **name that represents a value** in computer memory. ⇒ An assignment statement, also called **initialization**, is used to create a variable.
Ex: x = 10 where " = " takes the name of *assignment operator.*
Variables can be used in programming statements to represent the value they reference.
ATTENTION! Variables should never be placed in quotation marks, otherwise it will be considered as any string and will not indicate the corresponding value.
When writing variables, pay close attention to uppercase and lowercase ⇒ Python is **case-sensitive** = it distinguishes between lowercase and uppercase characters. (By convention, only lowercase letters are used when writing variables.)

In particular: THE NAMES OF THE VARIABLES
- must be **short and meaningful**;
- they must start with a **letter** or with an **underscore** (_);
- cannot contain spaces or special characters (#, @, €, $; §);
- cannot consist of Python **keywords**;
- it is preferable not to use accented characters.

Variables can **reference different values during the execution of a program** = can be **reassigned**: When you assign a value to a variable, it references the value until you assign it a different value. → Before updating a variable it is essential to initialize it, as otherwise it would generate an error.

Python provides **multiple** variable **assignment** (*or unpacking*): it allows you to act on multiple variables directly on the same line of code.

```
>>> a, b, z = 1, 2, 3
>>> a
1
>>> b
2
>>> z
3
```

Data types

When we use data in a programming language, it is necessary to take into account the *type of data* we are using to know what we can do: data management has specific rules.

[→ Some operations can only be performed with certain types of data, while others behave differently depending on the type of data they deal with.]

⇒ For this reason there is *typing*: a management of data that varies according to their type. → There are two types of *typing*:

- *static*: the programmer must explicitly declare the type of the variable before using it;
- *dynamic* (as in Python): it is the interpreter which, based on the value assigned to the variable, decides its type.

Data type	Name	Description	Examples
Integer	int	Integer of arbitrary size	112, 0, -158
Real number	float	Floating-point number	2.14, -159.1234
Boolean	bool	For true or false values	True, False
String	str	Used to represent text	'Python', 'Python3', 'Web 2.0',

To find out which data type a value belongs to, you can use the **type** function: it returns the type of the value that the variable refers to at that moment.

ATTENTION! In Python, currency symbols cannot be written to numeric values.

The numbers must be written in the simplest possible form: only with the decimal point (the thousands separator can be adjusted by the *format* function)

<u>Difference between using commas and using the + operator:</u>

```
>>> total=1500.50
>>> print('Totale: €',format(total,',.2f'))
Totale: € 1,500.50
>>> total=1500.50
>>> print('Totale: € '+format(total,',.2f'))
Totale: € 1,500.50
```

- when using the comma in the print function, the different arguments are directly returned separated by a space
- spaces are not inserted automatically in the chaining and must be managed manually.

ATTENTION! The concatenation operator works **only with strings**.

When performing a calculation between two operands, the data type of the result depends on the data type of the operands:
- two **int** operands -> **int** result;
- two operands **float** -> result **float**;
- one **int** and one **float** operand -> result **float**. ⇒ *expressions with mixed data types.*

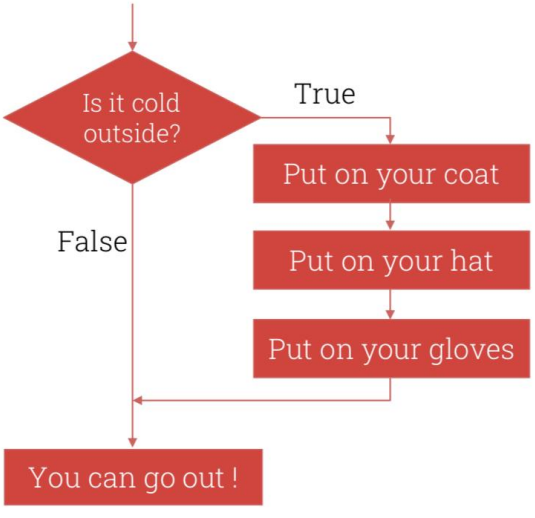
It is possible, using functions, **to convert a value from one data type to another:**
- **int** function: converts a given value into an integer;
  - It can also be used for decimal numbers, but it does not round the decimal part but truncates it.
- **float** function: converts integers and strings to floating point numbers;
- **str** function: convert the value of its argument to a string. → It can be useful when you want to display a result composed of numbers and strings on the screen.
  - If the conversion is impossible (because it is meaningless or not technically feasible) an **error** message appears.

Input function ⇒ Requests the user to enter data. → Once the user has typed in a value, the displayed output will be a string.

```
>>> name = input ("What's your name? ")
What's your name?  Bob
>>> print (name)
Bob
```

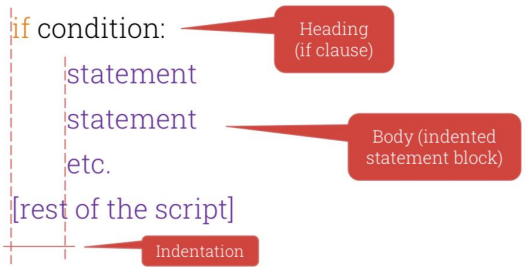
Sequence structure and decision structure:

Sequential structure = set of instructions that are executed in the order in which they are entered.
Decision structure = performs a certain action only when specific conditions are met.



There are different types of decision making structures:
- **simple** decision structure: single alternative execution;
- **alternative** execution: double alternative execution;
- **serial** condition.

Decision structures are implemented through the use of *conditional instructions* = instructions that allow you to divert the sequential flow of a program by introducing choices.
⇒ The conditional institutions are **if, else** and **elif**.



The **if statement** (*single alternative*)
- Header (**if clause**)
- Body (**indented statement block**): contains the statements to be executed if the condition is true;
- If the condition is false, the statement block is skipped and the program continues.

Boolean expressions → the conditions of the if statement are implemented by boolean expressions = expressions that can be either **true or false**.
⇒ Conditions use **comparison operators** and return True or False:

Important! Comparison operators apply *only* to variables and values of the same type.

The if-else statement
The *if – else* statement allows you to write a **double alternative** decision structure, which provides two possible execution paths, depending on whether the condition is true or false:

The *syntax of the function* provides:
- that the if and else conditions end with a colon;
- that the if clause and the else clause are indent-aligned;
- that the blocks of statements following the if clause and the else clause are indented uniformly.

The nested conditions => if -else and if – elif – else

If-else statements:
It can be used to test two conditions (with three possible outputs). →The syntax has two if clauses and two else clauses.

Instructions **if – elif -else**: it is used in case of **conditions in series**, that is when there are numerous cases and numerous possible exits.
With the **elif** statement it is possible to specify nested *if-else* blocks in a compact and readable way: it allows you to specify a block of statements for each of the possible cases.

```
if condition:
    statement
    statement
    etc.
else:
    statement
    statement
    etc.
[rest of the script]
```

```
if condition1:
    statements
else:
    if condition2:
        statements
    else:
        statements
[rest of the script]
```

```
if condition1:
    statements
elif condition2:
    statements
...
elif conditionN:
    statements
else:
    statements
[rest of the script]
```

Logical *or Boolean* **operators** = allow you to create complex boolean expressions, increasing the potential of conditional constructs:

Operator	Description
and	Returns **True** when all the arguments are true, otherwise it returns **False**
or	Returns **True** when at least one of the arguments is true, otherwise it returns **False**
not	Returns **True** if the argument is false and **False** when it is true

If statement with in ⇒ The if function test can also compare against a list of values or a range:

```
rain = input('Is it raining? (Yes or No) ')
temp = int(input('How many degrees are there outside? °C = '))

if rain == "Yes" and temp <=15:
    print('Put on your coat and take the umbrella!')

elif rain == "Yes" and temp >15:
    print('Take the umbrella!')

elif rain == "No" and temp <=15:
    print('Put on your coat!')

else:
    print('Have a nice walk!')
```

```
 if rain in ["Yes", "yes", "YES", "y", "Y"] and temp <=15:
     print('Put on your coat and take the umbrella!')
```

- the **in** keyword is required to use a list of values;
- to use a range of values you need the **range** function:
    - The range function returns an object that produces a sequence of integers, between the minimum value (included) and the maximum value (excluded) with increments corresponding to steps.

```
a = 12

if a in range(6,12):
    print('The value is OK!')
else:
    print('The value is not in the range')

The value is not in the range
>>>
```

The **pass statement** ⇒is a null operation: nothing happens when it is executed.

```
x = int(input('Enter the value of x: '))
if x > 0:
    pass    # add what to do with positive values
elif x < 0:
    pass    # add what to do with negative values
else:
    print('x is 0')
```

Iterative constructs

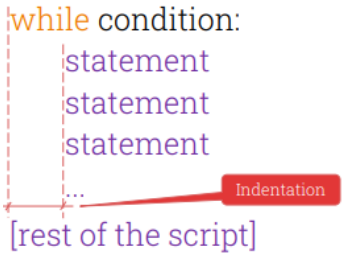
Iteration is the ability to repeatedly execute the same block of instructions. → Instead of duplicating the same sequence of instructions several times, in programming languages there is precisely the possibility of writing the operating code only once and inserting it in an *iterative structure*, also known as a **cycle** or **loop**.

Cycles can be:

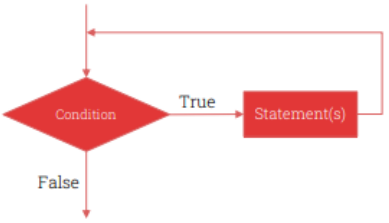
- **controlled by a condition** ⇒ use a *true/false* condition to control the number of repetitions. → **while loop**: repetition occurs as long as the condition is valid.
- **controlled by a counter** ⇒ the cycle is repeated a specified number of times. → **for loop**.

The while loop (loops controlled by a condition) → as long as the condition of the loop is true the block of instructions is executed; when the condition becomes false, the while loop is exited.
The while syntax requires:

- the while clause: it can assume the logical boolean value true or false and the closing colon;
- the block of instructions to be repeated.



A while loop can repeat an infinite number of times until a certain condition is met.



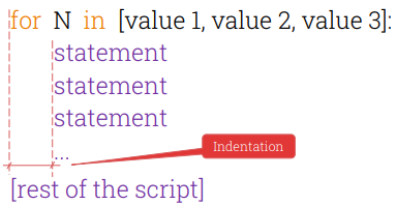
It may happen, in both *for* and *while* loops, that you need to abruptly terminate the loop when certain conditions occur. → **break statement**.
Or, when certain conditions occur, it is necessary to move on to the next iteration of the loop. → **continue statement**.

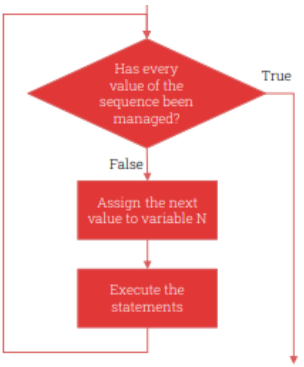
When the loop doesn't have an easily definable end ⇒ **the while loop True** = **infinite while loop**.
⇒ to end the loop *while True*: you need to add an if that contains the exit condition from the loop, and the statement executed when the condition of the if statement occurs must be **break** to exit the loop at that moment.

The for loop (loops controlled by a counter):
The for loop syntax consists of two parts:

- the for clause with the name of a **counter variable**, *in* and a closing colon;
- the block of statements that is executed on each iteration of the loop.

⇒ The counter variable is assigned the first value in the list, the instructions are executed and then the variable is assigned the following value and so on.



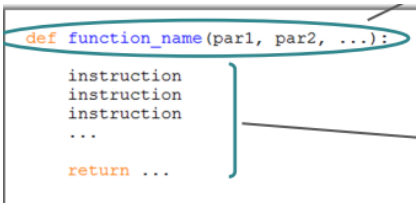


Functions and exceptions
In Python there are many functions that are immediately available, the **built-in functions**, and many more are available in the **standard library**; however, it is also possible **to create customized functions** to carry out tasks and operations according to your needs.
⇒ *Custom functions* are named blocks of code, which themselves can contain other functions, and **which are executed when explicitly called**.

The usefulness of being able to create ad hoc functions for your needs lies not only in the ability to calculate or perform an action, but also in the possibility of making a very complex program easier to read. → The advantages of a *modular program*:

- ability to reuse code
- greater simplicity of the code
- improvement in the test phase.

How to write and define a function in Python:



The first line is the *header*: it always starts with the keyword **def** followed by the function name and two parentheses and a colon. → Between the round brackets are the parameters that specify which arguments will be passed to the function: there are *mandatory or optional parameters*.

The subsequent lines of the definition constitute the *body*: it contains all the instructions necessary to carry out the requested operation. → The body must end with the **return statement** when it is necessary for the function to return a value (without the return the result is not stored).

The **function name** must respect some rules:

- the first character must be a letter or an underscore;
- keywords or spaces cannot be used;
- after the first letter you can use letters (upper and lower), numbers and underscores;
- for clarity, the name should make it clear what operation the function performs.

Every time you want to execute the function you need to **call it**: new_function() ⇒ specifying the parameters if required.
EXAMPLES:

```
def hello():
  print('Hello my friend, is everything ok?')
  print('Hope so!')
```

→

```
>>> hello()
Hello my friend, is everything ok?
Hope so!
```

```
def square(x):
    print('The square of', x, 'is:', x*x)
```

→

```
>>> square(5)
The square of 5 is: 25
```

Arguments of functions ⇒ Functions can have arguments: it is possible to pass inputs to the function.
To make it possible to pass data to the functions, **parameters** are used, i.e. variables that must be specified in round brackets.
You can set *mandatory parameters and optional parameters*:

- it is necessary to specify all the mandatory parameters first, followed by the optional ones;
- the default value that the parameter must assume if it is not specified when the function is called must be indicated in the optional parameters.

```
def function(par1, par2, par3=value, par4=value):
```

Productive functions and void functions: *the difference*

- ➔ A **productive function** is a group of statements that perform a specific task and, when finished, return a value to the statement that called it.
- ❖ Productive functions, therefore **always** end with the *return* statement.
- ➔ A **void function** is a function that performs a specific task, **but returns no value** when it terminates and is therefore *empty*.

Local variables and global variables:

- ➔ A **global variable** can be accessed from any statement of a program.
- ➔ A **local variable** is accessible only within its context (or scope) and therefore *only in the part of the program in which it was defined* = all the instructions that are outside the function cannot access the variable or use it.

The documentation string (docstring) ⇒ insertion of a few lines explaining the operations performed by the function or, more generally, a comment on the function.
⇒ It is NOT a mandatory operation, but it can be very useful to specify some features of the function.

```
def multiply(num1, num2, num3):
    ''' The function multiplies the three arguments and returns the result.\n \
        Arguments can be integer or decimal numbers'''
    return num1 * num2 * num3
```

```
>>> multiply(
          (num1, num2, num3)
          The function multiplies the three arguments and returns the result.
          Arguments can be integer or decimal numbers
```

The contents of the documentation string are shown in the *call tip* when calling the function in the shell.

Functions with loops and conditional expressions ⇒ In the body of the functions it is possible to use loops and conditional expressions to increase the possibilities of carrying out complex operations.

```
def sum_range(num):
  total = 0
  for n in range(1,num+1):
    total = total + n
  return total


def odd_even(num):
  count_odd = 0
  count_even = 0
  for n in range(1,num+1):
    if n % 2 == 0:
      count_even = count_even + 1
    else:
      count_odd = count_odd + 1
  print("Number of even numbers :",count_even)
  print("Number of odd numbers :",count_odd)
  return count_even
```

THE EXCEPTIONS ⇒ these are events triggered by errors of various kinds: if we expect them to occur, it is possible to write a code to manage them.

➢ When an exception is not handled it causes an **error** and the exit from the block of statements in which it occurs from the program.

There are different types of errors:

- **Syntax** errors: indicate that there is an error in writing the code. → **SyntaxError**

➢ The error message indicates where in the code the error is located, but does not specify how to fix it.

- **Runtime** errors: indicate that there is an error in the code even if the syntax is correct.

➢ shows an error message (*only in the shell*) and indicates the code that generates the error and specifies its cause.

- **Semantic** errors: occur when the program is executed without producing error messages, but the results are NOT the correct ones. → They require a re-read of the code or the use of the **debugger** for the resolution.

Exception handling ⇒ via the **try…except** statement:

```
try:
    a = int(input('Enter a number: '))
    b = int(input('Enter another number: '))
    print(a/b)

except ValueError:
    print('\nEnter integer numbers only!')
except ZeroDivisionError:
    print("\nOne of the two values is 0! \n \
    Please try again from the beginning with another number")
except:
    print('\nSomething's wrong: try again with other numbers!')
```

ValueError = inserting text instead of numbers, or decimal numbers;
ZeroDivisionError = inserting a zero in the denominator.
Debugging = research and removal of code errors. ⇒ occurs through a process of re-reading the code, debugging, modifying the code and re-running the program.

➢ There are special tools (e.g. Debugger).

The sequences

Sequences are objects that contain several data that are stored one after the other.
→ There are several types of sequences: **lists, strings and tuples**.
Sequence features:
- They are **iterable** objects = I can access them
- They can be **mutable**, they are only **lists** (once the sequence has been created I can modify it) or **immutable**, they are **strings and tuples** (they can no longer be modified after creation).
- **Index** = number that identifies the position of each value.

Operators of sequences:

Operator/function	Description
+	Concatenates/merges two sequences
*	Creates more copies of a sequence and merges them
in	Returns *True* if an element is found in a sequence, otherwise it returns *False*
len(seq)	Returns the number of elements of the sequence
max(seq) and min(seq)	Returns the largest/smallest value in a sequence consisting only of strings or numbers
sorted(seq)	Returns a new list with the elements sorted in ascending order
sum(seq)	Sums the elements of the sequence (numbers only)

Strings = sequence of characters (letters, numbers and symbols)
⟶ Since the string is immutable, the only way to change a string is to create a new string.

Lists = collection of data (called *elements*) of any type, enclosed in square brackets and separated by commas.
- Can be assigned to variables;
- An empty list is a list without elements => utility: I could create an empty list because then I might need to add it inside my program;
- A list can be **nested** inside another list;
- You can also create a list using the built-in **list** function.

Tuples (similar to list) = contain elements of any type, separated by commas and enclosed in parentheses (they are not required).
/⟶ you can use the *built-in* **tuple** function.

Sequence operators and functions: Applicable to strings, lists and tuples.

Indexing = allows you to access the individual elements of a sequence, using the syntax **sequence[index]**

- The index of the first element is equal to 0 and that of the last one is equal to the number of elements minus 1.

Slicing = function that can be used to extract a portion of the sequence.
⟶ **sequence[start_index : end_index : step] =** returns all elements of the sequence between the one with start index (**inclusive**) and the one with final index (**exclude**).

Example: (considering the string 'Computer viruses are an urban legend')

Command	Meaning	Result
MyQuote[6:12]	Select all the elements from the one with index 6 to the one with index 12 (not included)	`'er vir'`
MyQuote[:7]	Select all the elements from the first one up to the one with index 7 (not included)	`'Compute'`
MyQuote[3:20:2]	Selects an element every two starting from the one with index 3 up to the one with index 20 (not included)	`'ptrvrssae'`
MyQuote[:15:3]	Selects an element every three starting from the first one	`'Cpevu'`

If omitted **start_index**: elements starting from the first.
If omitted **end_index**: elements up to the last.
If both are omitted: entire sequence.

What are the methods?
Methods = special functions of strings, lists and tuples.
⟶ function syntax: **object.method(argument)**

⟶ **String methods**

Method	Description
.upper()	Returns a copy of the string with all alphabetic letters converted to uppercase. Any character that is already uppercase is not modified
.lower()	Returns a copy of the string with all alphabetic letters converted to lowercase. Any character that is already lowercase, is not modified
.find(sub)	Search in the string for the substring specified in the *sub* argument and returns the index of the first occurrence found
.startswith(prefix)	Returns *True* if the string starts with the string specified in the *prefix* argument, otherwise it returns *False*
.endswith(suffix)	Returns *True* if the string ends with the string specified in the *suffix* argument, otherwise it returns *False*
.count(sub)	Returns the number of occurrences of the substring *sub* in the string
.split()	Splits a string in words and returns a list of strings, considering the space character as the separator between the words
.join(iterable)	Returns a string obtained concatenating all the elements of an iterable object containing only strings

→ **Methods of lists**

Method	Description
.append(element)	Appends new elements to the list
.insert(index, element)	Inserts the desired element in the position specified by the *index* parameter
.remove(element)	Searches for the specified element in the list and removes the first occurrence
.pop([index])	Removes and returns the element with the specified index. If we omit the index, it removes the last element of the list
.index(element)	Searches for the specified element within the list and returns its index
.count(element)	Counts how many times a given element is found in the list
.sort()	Sorts the elements of the list in ascending order
.reverse()	Reverses the order of the elements of the list

→ **Tuple methods**

Method	Description
.index(element)	Searches for the specified element within the tuple and returns its index. If the element occurs more than one time, it returns the index of the first occurrence
.count(element)	Returns the number of occurrences of a specified element in a tuple

Dictionaries = object that contains a collection of data or elements.

→ Elements consist of two parts: **a key and a value.** ⇒ Keys are unique and can be any immutable data type. Values can be any mutable or immutable data type.

→ To create a dictionary it is necessary to write the elements separated by commas (,) **in braces { }**. Each element consists of a key followed by a colon (:) and a value.

→ Alternatively you can use the **dict** function, which creates a new dictionary with no elements.

→ Dictionaries are mutable objects.

To extract a value from a dictionary, the key associated with the value is used: **dictionary_name [key].**
To add key-value pairs: **dictionary_name[key] = value.**

Operations with dictionaries:
- **in** allows you to check the presence of a key
- **del** removes a key-value pair from a dictionary
- **len** returns the number of key-value pairs

Dictionary methods:

Method	Description
.get(key[,default])	Returns the value associated with the key specified in the *key* argument. If the specified key is not found in the dictionary, it returns the value *None* or the value specified in the *default* argument
.pop(key[,default])	Removes the key (and the associated value) specified in the key argument and returns the associated value. If the specified key is not found in the dictionary and a default value has not been set in the default argument, it returns a KeyError error
.popitem()	Removes the last key-value pair added to the dictionary and returns a tuple with two elements (the key and the value removed)
.items()	Creates a dict_items object consisting in a list of tuples, each one with two elements (the key-value pairs)
.keys()	Creates a dict_keys object consisting in a list in which each element is one of the keys of the dictionary
.values()	Creates a *dict_values* object consisting in a list in which each element is one of the values of the dictionary

Traversing dictionaries:

```
>>> address_book = { 'pippo': '555-123456',  'pluto': '555-654321' }
>>> for key in address_book:
        print(key)

>>> for key, value in address_book.items():
        print(key, value)
```

Object-Oriented-Approach: every 'variable/element' is actually an object
OOP is based on:
- encapsulation
- inheritance
- polymorphism

★ a ***class*** is the family of objects of a certain type, different libraries might have additional classes of objects ex. student
★ an ***instance***= ***object*** is a specific case of a class ex. Anna Bianchi, Li Wang…
★ ***attributes*** are the properties of objects ex. Name, ID number, Height…
★ ***methods*** are the functions performed with the object ex. give exams, change curriculum

Dir

can be used to view the classes of a module: >>> dir (turtle)
and the list of attributes and methods:
>>> dir (turtle.Turtle())

>>> bob = turtle.Turtle()
>>> dir (bob)

Call tip

to get help while coding, can call tip (Edit ot Ctrl + backslash) after creating an instance
>>> bob.

Create a class

- To create a class you need to give it a name (with first capital letter) and define its attributes and/or methods. parenthesis are not mandatory
- All instances of the class inherit attributes and methods of the class to which they belong, indented
- Attributes and methods inherited by individual instances can be assigned in the creation phase of the instance and/or changed at a later time

>>> class Person:

adding attributes

- They allow to customize each single object of the class (i.e. the instances)
- They can be Python built-in types (i.e.: int, float, list) or other objects
- They can be defined directly in the class (with an assignment, as shown here below), but they are usually defined with a special method called constructor method

Constructor method __init__

- like all methods, init is a function so it starts with def, but it is not productive
- It is the method that Python search in the class when it has to initialize the state of an object, if it is not explicit python uses the built in.
- It can include the definition of the attributes and of any other feature and option needed to the objects of the class, like: call of functions, opening of files, access to databases etc.
- The syntax follows the same rules used when defining a function, with the parameters (mandatory or optional) that are used to initialize the attributes, specified in parentheses
- There must be at least one parameter – conventionally named **self** – that represent the reference to the name of the instance that will be created

we can now **create objects of this class** that are initialized with the default attributes

assign attributes (*variable* or *constant)*

- To access a specific object, we must use the name of the related instance
- To define the value of an attribute we can use:

  InstanceName = ClassName(val_1, val_2, …)

  (only when the attribute has been defined as a parameter of the constructor method __init__)

  InstanceName.AttributeName = value

  (for attributes not defined as parameters. But it can also be used for those attributes defined as parameters in the __init__ method)

create methods for the class

- They are procedures/functions
- They can return a value thanks to the **return** statement
- They have at least one parameter: **self**
- They are invoked with the same notation used for attributes: Methods

InstanceName.MethodName(par_1, par_2, …)

Special method __str__

What if we just use the name of an instance as the argument of a print function?

- The printout will be the result of the special method **__str__**
- It has no arguments (besides **self**) and must end with the **return** statement

Inheritance

- parent class (superclass)
- child class (subclass)
  - inherits all data and methods of the parent class
  - adds more info and methods
  - overwrites methods

Access to files - open

To access a file in Python we use the built-in function **open**

- It opens a file and returns an object of type file to which is associated the content of the specific file that has been opened
- If the file cannot be opened, an error is raised
- Opening a file imply the creation of a variable to which the object should be assigned

Reading methods

The content of a file can be read entirely or line by line:

1. The **read** method reads the contents of the entire file
2. The **readline** and **readlines** methods read the content line by line

The modules

All programming languages can be enriched by extensions in order to execute specialized additional functionalities

- In Python, additional functionalities come as modules: these are files acting as containers, grouping useful functionalities by the topic they relate to
- Some are part of the basic Python installation, and we already met some examples as math, random and turtle
- Modules can be organized in libraries or «packages»
- Modules installed by default with Python make up the Python standard library
- There are also many custom modules available for the most diverse purposes

Import

Assuming a module is installed, in order to use it we need to import it into the program's running session.

Invoking a functionality of a module without having imported it, raise an error message:

The command we need is import, which also allows to import different modules at the same time:

N.B.: importing is valid only for the current session

Dir

The **dir** command shows the list of the loaded modules (the ones always available in Python, and those loaded in the current session):

The same command also allows listing the **functions and classes** offered from a specified module

To list all available modules: >>> help ('modules')

1. **os**

Standard module that provides a set of tools to make possible to interact with the Operating System, quite useful to inspect and manipulate files and folders. Some of its functions are:

os.listdir([path]) Returns a list with names of files and folders available in the folder specified as argument

os.path.join(path, filename) Returns the complete path of a file (*filename* argument) in the specified folder (*path* argument)

os.path.isfile(path) Returns *True* if the path is referring to a file, *False* otherwise

os.getlogin () Return the name of the current Windows user

os.getcwd() Return the path of the current working directory

os.chdir (path) Change the current working directory to **path**. (**path** should be written as a text string, e.g. 'C:\\Python')

os.rename(OldName, NewName) Renames files or directories from *OldName* to *NewName* (names must be strings, including the full path)

```
import os

def walk(path_arg):

    for strname in os.listdir(path_arg):
        path = os.path.join(path_arg, strname)
        if os.path.isfile(path):
            print('Found file:', path)
        else:
            print('Found folder:', path)


walk('C:\\')
```

2. random

Allows generating random numbers, with some useful functions:

random.random() Returns a random decimal number between 0 and 1

random.randint(min, max) Returns a random integer number between *min* and *max* (both inclusive)

random.choice([list]) Returns a random element from the list given as argument

random.randrange(min,max,step) Returns a random integer between *min* and *max* (this last excluded) with an increase of *step*

example:

In the following script the random module allows to chose randomly a student from a list created opening the file 'students.txt':

```
import random

student_list = []
student_file = open('students.txt')
student_row = student_file.readline()

while student_row != '':
    student_list.append(student_row)
    student_row = student_file.readline()

print("Now it's the turn of", random.choice(student_list))

student_file.close()
```

3. webbrowser

It is a Web-browser controller that provides a high-level interface to allow displaying Web-based documents to users.

One of the most relevant functions is:

webbrowser.open(URL, new=0)

Display the web page available at **url** using the default browser. If **new** is **0** the **url** is opened in the **same browser window**, if possible. If **new** is **1** a **new browser window** is opened, if possible. If **new is 2 a new browser page** ("tab") is opened, if possible

example:

In the following script the wiki function uses the webbrowser module to open in the default browser the Wikipedia page of the keyword passed as argument when the function is called

```
import webbrowser

MyQuery = ''
MyKW = input('Type the keyword you want to search: ')

while MyKW != '':
    MyQuery = MyQuery + MyKW + '+'
    MyKW = input('Do you want to enter another keyword?\n\
    (Type the new keyword or hit Enter key to launch the search): ')

webbrowser.open('https://www.google.com/search?q='+MyQuery)
```

Custom modules

Specialized modules, usually intended for niche audiences, that must be installed by the user.
Where to find them?

PyPI - **the Python Package Index** is the official site collecting and categorizing most of the available modules There are many other sources offering information and download of modules. For instance **UsefulModules** contains a list of the most popular Python modules classified by topic, mainly aimed at a Python beginners

Installing custom modules

To use a custom module it is necessary to **download** and **install** it following the instructions, or simply running the command *pip install modulename* from the command line of the operating system:

- Write **cmd** in the search box of the Windows menu (in MacOS, open the Terminal application)
- In the command line, we write **pip install *modulename***

openpyxl module

Non standard Python module for reading and writing Excel files.
Some useful functions:

★ **MyNewXLfile = openpyxl.Workbook ()**

Creates the instance

★ **MyNewXLfile** of the class of objects Excel workbook (Note: a new empty workbook has always one single worksheet)

★ **MyNewXLfile.save ('MyFile.xlsx')**

Save the Excel workbook object

```
import openpyxl

MyWorkbook = openpyxl.load_workbook('companies.xlsx')
MyWorksheet = MyWorkbook['Feb']
idx = 7
companies_list = []

while MyWorksheet.cell(row=idx, column=3).value != None:
    companies_list.append(MyWorksheet.cell(row=idx, column=3).value)
    idx = idx + 1

print('\nCompanies:\n')
for company in companies_list:
    print(company)
```

★ **MyNewXLfile** as MyFile.xlsx on the active path

★ **MyNewXLfile.active.title = 'MySheetName'**

Rename the active worksheet i**n MyNewXLfile** workbook as MySheetName

★ **MyNewXLfile**.create_sheet ('MyNewSheet')