COMPUTER SCIENCE : PYTHON

HANDOUT

EDITION A.Y. 2019 – 2020

Written and edited by:
Benedetta Magni, Francesco Citti and Irene Miraglia

This handout has been written by students with no intention to substitute the University official materials. Its purpose is to be an instrument useful to the exam preparation, but it does not give a total knowledge about the program of the course it is related to, as the materials of the university website or professors.

INTRODUCTION TO PROGRAMMING

Programming consists in instructing a machine to realize a given task that can both be simple or complex. Hereby the definition of program: "A program is a set of coded instructions that enables a machine, especially a computer, to perform a set of desired sequence of operations". Nowadays, such competencies are also useful for web marketing, big data analysis and many others.

DATA STORAGE AND HOW A PROGRAM WORKS

Data storage occurs through 3 components of the hardware: CPU (Central Processing Unit, that carries out operations), memory (ROM, RAM or Mass Storage) and other input/output devices (such as keyboard); these components communicate with each other through BUS channels. The CPU only understand machine language (given by 0 and 1), it is then necessary to use high level programming languages (such as Python) whose instructions are translated into this binary language. It is possible to translate multiple information in 0 and 1 sequences (such as numbers, letters, pixel colors, sounds, etc.).
To run a program, it is necessary to translate the source code. This operation can be carried out in two ways:

- **Compilation,** the program produces and executable file (ex: file.exe) that can be directly used. The advantage if this option is that it is a quick process, however the fact of having translated it entirely makes it more difficult to identify mistakes;
- **Translation,** the program is translated and executed one single instruction at a time. On the one hand it makes the execution slower but on the other, it is easier to pinpoint possible mistakes.

LANGUAGE EVOLUTION

At the beginning, it was necessary to program using machine language, but it being quite complex, in the 40s Assembly (considered low level) was introduced. This language still needed a "translator": Assembler. In the 50s the first high level languages appeared: COBOL then C, Java, C++, Python, etc. These programming languages use a syntax very similar to English. They can be split between:

- **Declarative**, describing reality and declaring a requested objective;
- **Procedural**, defining algorithms that lead to an objective.

Python can be considered both procedural and declarative.

DEVELOPMENT CYCLE, ALGORITHM AND FLOWCHARTS

When developing a program, we start from some **requisites** that the program needs to match; we then move to **designing**, in which we indicate roughly what the program will have to do. Afterwards it will be necessary to **implement** this project in the real program that will be eventually **tested**, to delete some potential mistakes (**debugging**).
In the design and implementation phases we will deal with the **algorithm**, defined as a finite sequence of coded actions leading to the solution to a problem. The algorithm's instructions need to be finite, materially executable, not ambiguous and carry out results.
The algorithms can be expressed as **flowcharts**, using geometric elements with precise meaning: ovals, beginning and end of the algorithm; rhomboid, input and output data; rectangles, elementary actions; rhombus, decisions elements and arrows indicating direction of execution.

INTRODUCTION TO PYTHON

WHAT IS PYTHON?

Python is a high-level interpreted, interactive and object-oriented programming language.
It was introduced in 1991 and has been developing ever since. New versions appeared contributing to increasing progressively Python's popularity, also given it is simple to use compared to other languages. Python, being an open source language, continuously completed by several libraries and compatible with most common operating systems, spread easily.
We mentioned that Python is an interactive language, meaning that it is possible to write several instructions in its command prompt (shell); it is also object-oriented, as the solution to the problem is not simply expressed as a sequence of instructions but actually in terms of objects and their attributes.
Finally, a dynamic typing of variables is essential in Python: indeed, a fundamental element in Python are variables containing data, values coming from operations or computations executed by the program. To use a variable, it is sufficient to define the type (string, integer number, etc.). Then, it is possible to carry out several operations with the above-defined variable, always keeping in mind that some operations are not possible on every type of variable. (Conversion functions exists.)

IDLE: EDITOR AND SHELL

IDLE (Integrated Development and Learning) is the programming environment that comes along when installing Python, although it is possible to use it in other environments. It is composed of the **shell** and the **editor**.

- The shell interprets Python commands, it is used interactively and allows to carry out commands one by one once typed. This condition of use is useful when we try brief commands before writing the real program in the editor.
    - The shell is the first window that opens once the program is launched; it shows, at the top, an opening message followed by the prompt (>>>) notifying that the program is ready to receive new instructions.
    - If some erroneous instructions are inserted in the shell, an error message is immediately shown.
    - In the shell, it is not possible to modify or erase the written command, it has to be re-written. Moreover, it is not possible to "clear" the shell, it has to be restarted. (Shell -> Restart or Crt+F6)
    - Finally, data, once you quit the program, is not saved.
- The editor that you access through File -> New file or CTRL+N, is only used in script modality. Contrary to the shell, it can save files with the extension .py so that they can be modified and run. While in the shell, instructions are executed any time you press enter, in the editor programs are written entirely and can then by run (Run -> Run Module or F5).

Selecting Options -> Configure IDLE, you have access to the different options of IDLE, that can be modified at whatever moment and include Keys (shortcuts), that allow you to work more easily. An important shortcut is the history-previous (ALT+p), that recalls the last inserted command.
In the Highlights, it is possible to see to what corresponds each color used during programming. Python has indeed assigned colors to the different types of elements (appearing automatically as you write):

- Purple: built-in functions (ex: print);
- Green: strings (between " ") but also comments inserted out of the coding lines;
- Blue: output of the instruction or name of function (user-defined and class functions);
- Red: errors but also comments between lines (preceded by a #)
- Orange: keywords (ex: if, else, etc.)

WRITE AND RUN A PROGRAM

To write a new program it is necessary to open the editor. It is possible to maintain both the editor and shell windows open. Once the code has been written in the editor, we must save the file (File -> Save as or File -> Save).
Moreover, we can also save a copy of the file keeping the original open (File -> Save Copy as).
As soon as the file is saved, it will be possible to run the program (Run -> Run Module) or control the syntax, which will indicate possible mistakes in coding (Run -> Check Module).
If we decide to run the program without doing the check first, a message "Invalid syntax" will be shown in case of error and the position of the error will be highlighted in red in the editor before the program is run in the shell.

To open a .py file, you shouldn't double click the file, regardless of whether you want to run or simply open it. To open the file, it is needed to right click on it and select Open -> Edit with IDLE.

FIRST NOTION ON PYTHON SYNTAX

Every programming language has its own syntax; Python's is formed by keywords, operations, punctuation and other elements needed for the execution of instructions.
The syntactic rules establish how elements should be "disposed" in the lines in order to create some precise instructions.

Python is a flexible program regarding spaces, it is indeed possible to insert or not insert spaces between elements since the program will recognize them anyways. For example, it is possible to write 3+3 or 3 + 3 and the sum will still be executed; it is still advised to use spaces for an easier reading or at least to be coherent (either always put spaces or never).

Python is however pretty rigid concerning the so-called indentation, an indent before entering some functions or text. It is necessary for some instruction as for example if.

It is important to underline that Python keywords have a precise meaning and cannot be used for other scope.
Here are the keywords:

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

Beside the keywords, some mathematical operators are also present in Python: +, -, *, /, //, %, ** (explained later).

In Python the sole operators that can be used with text strings are + (which concatenates different strings) and * (that repeats the same string several times). It is not possible, however, to execute operations between strings even if their content seems numerical.

When using text strings, as in the print function, the string should be written between single quote marks (' ') or in inverted commas (" "), according to whether in the string it is necessary to use one or the other (the one that is not in the text). In case of strings that extend over more lines, triple quote marks or triple inverted commas should be used.

Last but not least regarding strings, if the string is too long and you wish to write on a new line without having a new line in the final output, you should then insert, before starting on a new line, a backslash \.

TYPES OF MISTAKES

Two types of error can occur:

- Syntax errors: in this case the message SyntaxError: invalid syntax will be shown, highlighting a syntax mistake.
- Error during the execution of the program: when the syntax is correct, but an execution problem occurs. In this case the last line of the displayed message contains the explanation for the problem as well as the type of error (NameError, TypeError, etc.)

COMMENTS AND DOCSTRINGS

In Python it is possible to document the actions and add comments, or in other words brief notes explaining some steps.
If the comment is inserted in the same line of the piece of information we are writing, it has to be preceded by a # and it will automatically be colored in red.
If the comment is written on a new line on its own, then beside the #, it must also be written between triple quote marks or triple inverted commas. In this case we talk about docstrings.

ESCAPE CODE

The escape codes (escape sequence or escape characters) are commands preceded by a backslash (\), positioned within a string.

BUILT-IN FUNCTIONS

Those are the "default" functions already existing in Python.
The name of a function will appear purple while the string will be green. The other elements of the function (ex. Variables) will be black.
When writing the name of the function it is necessary to type it in lower-case letters as Python distinguishes upper and lower-case.
Every time the name of the function is written and the parenthesis open, a yellow square will show the so-called "tip" containing the syntax that should be used for the corresponding function, meaning the different parameters that can be inserted, separated by a comma. It is possible to modify in settings the tip call using Edit -> Show Call Tip if they are not visible.
An example of a build-in function is print, allowing to display on the screen a string of text. The fundamental parameters for this function are: value (string or elements we want to display), sep=' ', indicating with what we want to separate the various elements, and end=\n.

HELP AND OTHER SUPPORTS

Python includes support tools that allow to search information in case problems or doubts occur.
One of these is the help function, that you use in the shell. In the interactive modality it is then possible to search for information on any type of data, on a function or on other elements of Python. We can also use the function help over itself typing help(help) and clicking on enter.
If we only wrote help(), we enter in an interactive modality in which it is possible to view information on several elements, one after the other.

Other types of support are offered through the program Python 3.x Manuals, including a navigable version of the documentation about Python. More information can be found on the official website www.python.org.

OPERATIONS AND OUTPUT

MATHEMATICAL OPERATORS AND CALCULATION

In calculation on Python we have several mathematical operators.

Symbol	Operation	Description	Example
+	Addition	Adds two numbers	>>> 12 + 4 16
-	Subtraction	Subtracts two numbers	>>> 45 –15 30
*	Multiplication	Multiplies two numbers	>>> 14 * 3 42
/	Division	Divides two numbers and returns a floating-point number	>>> 56 / 3 18.6666666666666667
//	Floor division (or integer division)	Divides two numbers and rounds down to an integer (*)	>>> 56 / 3 18
%	Modulus	Divides two integers and returns the remainder of the division	>>>16 % 3 1
**	Exponential	Elevates a number to a power	>>>5 ** 2 25

(*) when the result is negative it is rounded down to the nearest whole number.

ORDER OF OPERATIONS

The order of operations when computing a result follows the ordinary order of algebra thus parenthesis, exponentiation, multiplication, division, addition and subtraction (**PEMDAS**). Ex. (10-2)**7**2=$(8^7)^2 = 8^{14}$.
Moreover, in Python it is not possible to omit parenthesis or sign, ex: 2xy shall be written 2*x*y.

FORMAT FUNCTION

To modify the way numbers are viewed in Python we use the format function. For example if we wanted to format 10/3=3.33333333 we would write: format(10/3, '.2f'), and our output would be 3.33. The function is composed of two parts: the first is the number to be formatted, the second is the format specifier, it is indicated between single quote marks; in the example we communicate to Python to format to the second decimal number, **f = float**.
There are other formatting styles such as '.%' that will be used to format a percentage, or ',.2f' in which on top of considering 2 decimal numbers, thousands will also be separated by a comma.

OTHER MATHEMATICAL FUNCTIONS

sum = sum of the numbers between parenthesis	>>>sum((4, 5, 6)) 15
pow = elevation to a power	>>>pow(4, 2) 16
abs = absolute value	>>>abs(-8) 8
max e min = compute maximum or minimum of the list in parenthesis	>>>max(1, 5, 7) 7
round = rounds a float number at the figure indicated as second element	>>>round(1.55, 1) 1.6

These functions constitute the Python **built-in** function. Beside those, there are the **standard library** functions (already installed but needing to be "recalled" when we want to use them in a new file) and the **external library** functions.
An example of a standard library function is the **random.randint(a,b)** function, that belongs to the "random" library, which will produce a random output between two numbers a and b.

```
>>> import random
>>> random.randint(0,7)
5
```

VARIABLES AND TYPES OF DATA

VARIABLES AND ASSIGNEMENT STATEMENTS

Python uses these so-called variables, meaning names that represent a value contained in the memory of the computer. To use a variable, it is first of all necessary to initialize it, giving it a name and assigning it an initial value that will be stored in the memory but will still be modifiable.
The typical structure would be:
>>> name = value
In which the '=' sign is called **assignment operator**.

Variables are very important as they can be used inside the functions (with no need of quote marks) to facilitate typing the program.
We must however remember that if we use a variable that has not yet been assigned, the program will show an error (ex. Using the variable "name" without assigning it will result in: NameError: name "name" is not defined).

VARIABLES NAMES

Some rules and conventions exist when naming variables.
First of all, it is advised to use meaningful and short names, not single letters that may create confusion.
Each variable must begin with a lowercase letter or a _and a number, moreover it is not possible to use special characters (@, !, &, etc.).
If the name is composed of more words, as a convention we use the underscore as separator (it is also possible to have each following word starting with an uppercase letter but reading would be less easy).
>>> surname_name = 'Wilson James'
Or
>>> surnameName = 'Wilson James'
Finally, using lowercase letter is advised since Python is a case sensitive language.
It is also necessary to pay attention to use function names as names of variables : when, for example, we use 'print' as a variable name, the print function loses its usefulness and there will be an error TypeError:'int'

object is not callable. In order to reset the function to its original functionality, it will be necessary to restart the shell.

REASSIGNMENT

When writing the program, it is possible to reassign a value to a variable through the reassignment process.
>>> a=1
We can hereafter write:
>>> a = a + 1
This way the value taken before is deleted from the memory.

UNPACKING

It is possible to realize the so-called unpacking, or multiple assignment, in which multiple variables can be initialized all at once.

DATA TYPES

In Python, several types of data can be used, each managed by the program in a different way.
Coding is dynamic, meaning that the program is capable of understanding which type of variable is considered, without the need for the operator to specify it.
Types of data in Python are:

Type	Name	Description	Example
Integer	int	Integer of arbitrary size, negative or positive	100, 0, -100
Real number	float	Floating-point number	100.9 , 0.1 , -100.5
Boolean	bool	For true or false values	True, False
String	str	Used to represent text	'Python' 'Web 2.0'

Lists also exist, they are a complex type of data (ex: x =[1, 2, 3]).
To recognize the type of data that is being used, it is possible to use the function **type** that displays as a result the type of data of the value in question.
>>> type('Python3')
<class 'str'>
It is important to remember that there doesn't exist an "evaluate" function, that means that plugging '$100.90' in the type function would result in a string. In order to display the value we must use the format function:

```
>>> total_amount = 1000.90
>>> print('Total amount: $'+format(total_amount,',.2f'))
Total amount: $1,000.90
```

N.B. Using the + instead of the comma doesn't display a space between $ and the number.

EXPRESSIONS WITH MIXED DATA

Calculations executed between two different data types will have different outcomes:

- Operations between two int give an int
- Operations between two float give a float
- Operations between an int and a float give a float

CONVERSION BETWEEN DATA TYPES

It is sometimes necessary to convert a value from a type to another to carry out some operations, such as when a function displays a string when we need a float or int.

INT FUNCTION

The int function takes a value and converts it, if possible into an integer.

```
>>> price = '100'
>>> int (price)

100
```
In this way we only view the integer value corresponding, but we haven't converted the the type.
If we want to convert from str to int:
```
>>> price = '100'
>>> price_int =  int (price)
>>> price_int
100
```
If the conversion is not possible, the error ValueError : invalid literal for int() with base 10: 'inserted string' will appear.
If we convert a float in int, the numbers after the decimal separators will be cut off and not rounded.

FLOAT FUNCTION

The float function converts integers and strings (if possible) in values with decimals.
```
>>> float (10)
10.0
```

STR FUNCTION

The Str function converts the value in a text string. It is very useful when we want do view results composed both by numbers and strings.
```
>>> profit = 100
>>> '€'+profit
```
An error would appear, in this case it is first needed to convert the numerical value in a string.
```
>>> profit = 100
>>> profit_str = str(profit)
>>> '€'+profit_str
'€100'
```

THE INPUT FUNCTION

The input function allows the user to insert data from the keyboard, that will then be used in the program. The function is frequently used along with the variables, so that what is typed can be used in other functions as a variable.
```
>>> variable = input(prompt)
```
Where prompt corresponds to what is asked to the user.
The type displayed is always a string even if it is a numerical value, indeed if we wanted to increment the value, the program would show an error:
```
>>> age = input('How old are you')
How old are you? 20
>>> age = age +1
```
You will then need to write
```
>>> age = int(input('How old are you?')
```

DECISION-MAKING STRUCTURES AND BOOLEAN EXPRESSIONS

THE IF STATEMENT

Up to now we only studied **sequential structure**, or in other words instructions executed one after the other. However there are cases in which the program, based on the condition or on the value of a parameter needs to apply different choices. We talked about a **decision-making structure**.

If is a conditional instruction and is written this way:

```
if condition:
    statement
    statement
```

The if is followed by a condition which, if true, the following instructions are executed. If false, the statement block is skipped, and the program continues.

BOOLEAN EXPRESSIONS

The conditions in the **if** instruction are Boolean expressions, that can display True or False (N.B. the uppercases are necessary) according to whether they are true or false. These values are bool data types, not strings.

Operator	Description	Example
==	Equal to	>>> 3 == 3 True
!=	Different from	>>> 3 != 3 False
>	Greater than	>>> 7 > 5 True
>=	Greater than or equal to	>>> 8 >= 9 False
<	Less than	>>> 8 < 10 True
<=	Less than or equal to	>>> 5 <= 1 False

N.B. "=" is an assignment operator whereas "==" is a comparison operator.
Furthermore, the comparison operators can be applied between variables of the same type.

```
x = int(input('Enter the value of x: '))
if x > 0:
    print('x is positive')
```

In this case, if x > 0 is true, the instruction print will ne executed, if it if false, the block is skipped and the program ends with nothing more.

IF-ELSE STATEMENT

The if-else statement is a **double-alternative** decision-making structure.
There are two possible execution based on whether the condition after the if is true or false.

```
if condition:
    statement
    statement
else:
    statement
    statement
```

Ex:

```
x = int(input('Enter the value of x: '))
if x > 0:
    print('x is positive')
else:
    print('x is negative or 0')
```

IF-ELIF-ELSE STATEMENT

This statement works the same as the if-else statement, with one or more elif command between.
The elif statement can be seen as a set of else-if; this way a **multiple-alternative** decision-making structure. We talk about chained conditionals.

```
if condition1:
    statements
elif condition2:
    statements
...
elif conditionN:
    statements
else:
    statements
```

in

LOGICAL OPERATORS: AND, OR, NOT

Expression	Meaning
x > y and a > b	x is greater than y and at the same time a is greater than b?
x == y or v != z	x is equal to a y or v is different from z?
not (a >= b)	Is the expression a >= b false?

N.B. There exist the **pass** operator, it is a null operation which is used to let functions on hold.
Ex:

```
if x < 0:
    pass    # add what to do with negative values
```

ITERATIVE CONSTRUCT

LOOPS

Loops are iterative structures that are used to execute the same block of commands several times. There are two categories of groups:

- Loops controlled by a condition
- Loops controlled by a counter

They differ in the way in which is established the number of times the operation should be repeated.

THE WHILE LOOP

The **while** statement enables the creation of a loop controlled by a condition and composed by:

- The **while** clause, with a condition that takes Boolean values (True/False) and end with ":"
- An instruction or a block of instructions (that have to be indented)

```
while condition:
    statement
    statement
```

This means that, until the starting condition is true, the block of instruction will be executed. When the condition is no longer true, we exit the while loop and we move on to the possible following instructions.

Ex:

```
keep_going = 'y'

while keep_going == 'y':
    input('\nEnter the student ID number: ')
    input('Enter the last name: ')
    input('Enter the name: ')
    keep_going = input('\nDo you want to enter data for another student? (y/n) ')
```

Output:

```
Enter the student ID number: 3201224
Enter the last name: Smith
Enter the name: John

Do you want to enter data for another student? (y/n) y

Enter the student ID number: 3215472
Enter the last name: Doe
Enter the name: Jane
```

If the loop doesn't have an easily definable end, it is possible to use a while True instead if while, with which it is not necessary to initialize the value before inserting it in the loop.
In whileTrue it is necessary, however, to use the keyword if, whose condition is the break keyword (allows to exit the loop).
Ex:

```
while True:
    n = input("Please enter 'hello!': ")
    if n == 'hello!':
        break
```

THE FOR LOOP

The **for** loop, is a cycle controlled by a counter. The instructions are repeated a given number of times.
It is composed by:

- A clause with the name of a counter variable, the word or range and ":" closing punctuation.
- An instruction or block of instructions that must be indented

To the counter variable is given the first value of the list or of the range (whether it is in or range). The number of executions is therefore limited.
Ex:

```
for num in [1, 2, 3, 4, 5]:
    print(num)
```

THE RANGE FUNCTION

The range function is another Python built-in function that has this syntax:
>>> range (start, stop [,step])
Start refers to the starting point, the integer number from which we start, stop is the ending point (also an integer). Step indicates the possible intervals between a number and the next that the function can show (default step is 1).
If the start is omitted, the function will start counting at 0. **IMPORTANT:** the stop is not included in the count: if we write range(1,5), 5 is excluded.
Ex:
range(1,5) -> 1, 2, 3, 4
range(5) -> 0, 1, 2, 3, 4
range(1,5,2) -> 1, 3
range(5,1,-1) -> 5, 4, 3, 2

If we insert a float number inside the range, the program will show an error TypeError:'float' object cannot be interpreted as an integer.

NESTED LOOPS

It is possible to create nested loops by inserting for and while inside other loops.
We can for example insert a for loop in another for loop, or a for loop in a while loop, or a while loop inside another while loop.
Ex:

```
for a in range (1,4):
        print('****', a)
        for b in range (3, 0, -1):
                print('--------', b)
print('End')
```

BREAK AND CONTINUE STATEMENTS

The break and continue statements are often inserted inside the loops.
The break statement is used when we wish to interrupt the execution of a loop by forcing the interruption. Continue is used when we are waiting to complete the writing of instructions, but we want to avoid an interruption of the program. During the execution, the program will go on instead of stopping.

FUNCTIONS

Beside the Python built-in functions, it is possible to create and personalize some others, at which a name is assigned in the program itself. The use of personalized functions allows to build modular programs, which have many advantages: possibility of reusing a code, more simple code and better testing phase.

DEFINING AND CALLING A FUNCTION

A function in Python is defined this way:

- Write "**def**".
- **Name of the function**, that has to respect some rules (as in the variable names): no spaces, first letter can neither be a number nor a Python keyword (return, while, etc.). The name should also respect the content of the function.
- We write between brackets the **parameters** that indicate the input arguments that will be used for the instructions of the function. We then put "**:**".
- We indent instruction.

Ex:

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

There can be functions that do not need parameters, in this case parenthesis are still written but left empty.

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

To "call" a function it is necessary to simply write in the shell the name of the function and parameters (if necessary):
>>> function_name (parameter)
N.B. If one if the parameters is a string it has to be put between single quote marks.

FUNCTION ARGUMENTS

How to use the parameters:
def subtract (a, b) :
print ("The result of the subtraction is: ", a-b)
The parameters are divided by commas and are used when defining a function, when num1 and num2 will have values (numeric in this case), they will be recalled by the function and will become arguments in the function itself.
>>> a = 9
>>> b = 4
The result of the subtraction is: 5

There can be **mandatory** and **optional** parameters, the firsts are necessary while the seconds, if not defined, have a **default value**.

def subtract (a, b = 4)
print ("The result of the subtraction is: ", a – b)

In this case, if we didn't assign a value to b, the function would start anyways considering b = 4.

In Python, parameters are considered in order:
>>> subtract (7, 4)
In this case a = 7, b = 4

It is possible to choose anyways which parameter will take which value:
>>> subtract (b = 5; a=7)
-2

PRODUCTIVE FUNCTIONS

If we insert the **return** instruction at the end of a function, the result of the function is saved in the memory and can be recalled, we talk about **productive** function:

```
def compute (a , b, c):
    result = a*b/c
    return result
```

```
>>> compute (3 , 4, 5)
2.4
```

If the function is not productive (every function before this part), it is called **void**, as it doesn't return any value to the instruction that called it.
We can verify in the shell whether the value was saved thanks to the return function or whether the function is void.

LOCAL AND GLOBAL VARIABLES AND DOCSTRING

Local variables are defined inside a function and are therefore useable only by the instructions in that same function. The global variables are useable by any function and instruction of the program.

```
def calc(x, y):
  temp1 = (x + y) / y**2
  temp2 = x**2 - y**2
  return temp1 - temp2
```

```
w = 100

def calc(x, y):
  temp1 = (x + w) / y**2
  temp2 = (x**2 - y**2) * w
  return temp1 - temp2
```

While w is a global variable, temp1 is a local variable created to make calculations easier and make the code more readable.

It is also possible to insert a comment by defining a **docstring** in order to explain or give more information about the function. The docstring should be enclosed in triple quote marks, and immediately following by 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
```

FUNCTIONS WITH LOOPS

Functions can contain loops or conditional structures (if, elif, else). There are no difference in the functioning compared to the loops already seen. Ex:

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

```
def discount(quant, price):
  if quant > 100:
    total = quant * price * (1 - 0.4)
  elif quant > 50:
    total = quant * price * (1 - 0.2)
  else:
    total = quant * price
  return total
```

STRINGS AND LISTS

SEQUENCES

A sequence is an object containing several elements. The most common forms of sequences are strings and lists. They have both differences and similarities: for example, both for lists and strings we can be use several functions and operations.

STRINGS

A text string is an alphanumeric sequence of characters. To indicate the start and the end of a string we use (') or (''). Strings are unchangeable. Indeed, to edit the content of a string assigned to a variable it is necessary to assign to it a new correct string, or to create a new variable, different from the first.

OPERATIONS ON STRINGS

In Python it is possible to carry out operation with strings, among which those using mathematical operators.

Operator	Operation
+	Concatenates several strings without leaving spaces. It is not possible to concatenate different types of data. It can only be used inside functions.
*	Creates a string that repeats the initial string n times.

in	Returns True if the string is contained in another, if not it returns False.
not in	Works in the opposite way of "in"
is	Returns True if a string is equal to another, False if not
is not	Opposite of "is"

Examples:

```
>>> text = 'every' + 'where'
'everywhere'
```

```
>>> 'Python'*2
'PythonPython'
```

```
>>> x='London is the capital'
>>> 'capital' in x
True
```

A list is a collection of different data, each of them called element. The various elements of a list are enclosed in square brackets and separated by commas. Lists can contain different types of date, thus not needing to be uniform. If we wanted to view the content of a list we could use the command print(list).
It is also possible to create a list:

```
>>> text=list('Milan')
>>> print(text)
['M','i','l','a','n','o']
```

OPERATIONS ON LISTS

As for strings, it is possible to use operators on strings.

Operator	Operation
+	Joins several lists, even if we have different types of data
*	Creates a new list being the initial list repeated.
in	Returns True is the element is contained in the list, False if not
not in	Opposite of 'in'

INDEXING

Indexing is a way to access to single elements of strings or lists. Indeed, each element has a determined position in a sequence. It is important to note that if the count starts from left we'll have positive numbers (starting from 0), if it starts from the right, then we'll have negative numbers (starting from -1).
Ex:

```
>>> capital_cities = ['Rome', 'Paris', 'London']
>>> capital_cities [2]
'London'
```

N.B. Rome's index is 0, Paris is 1, London is 2.

SLICING

Slicing allows to select a precise part of a string or list, using index.

```
>>> sequence[initial_index:final_index]
```

We must remember that, while the initial index (letter of the string/element of the list) is included, the final index is excluded by the selection.
If the initial index is omitted, the slicing will start from index 0. If the final index is not specified, the slicing will carry on until the end of the sequence.
We can also introduce the so-called **step**, in case we want to select equidistant positions (think about the **range** function step).
It is also possible to carry out the slicing with negative index. In this case the step must also be negative. If we wished to use a negative step with positive indexes, we would need to swap the first index with the last one.
N.B. Since lists are variable, it is possible to use the slicing to modify one or more elements of the list.

```
>>> text = 'PYTHON'
>>> text [2:5]
'THO'
>>> text [:4:2]
'PT'
```

```
>>> text [:]
'PYTHON'
```

STRING FUNCTIONS AND METHODS

Some functions operating on strings require, in their syntax, that at least one of their arguments is a text string or a variable containing a text string. Here are the most common:

Function	Description
len()	Returns the length of the string, counting the characters. Often used in loops within the range function.
min()	Returns the minimum value. If we consider letters, the smallest is 'a'. It distinguishes between lower and uppercase letters.
max()	Returns the maximum value. If we consider letters, the largest is 'z'. It distinguishes between lower and uppercase letters.

Methods are functions linked to a precise object:
String.method(arguments)

Method	Description
.upper()	Returns the string in uppercase letters.
.lower()	Returns the string in lowercase letters.
.capitalize()	Returns the string with the first letter uppercased and the others lowercased.
.strip()	Returns the string deleting initial and final spaces.
.find(sub)	Searchs in the string the specified substring and returns the index of the first occurrence found. If no substring is found, it displays -1.
.replace(old,new)	Returns the string in which all substring occurrences have been replaced with the new.
.startswith(prefix)	Returns True, if the prefix is found at the beginning, False if not.
.endswith(subfix)	Returns True if the subfix is found at the end, False if not
.count(sub)	Returns the number of times in which the searched substring appears in the string.
.split(iterable)	Splits strings at each predefined space and returns a list of strings. It is possible to insert a different separator as argument. **maxsplit** can also be added as an argument, indicating the maximum number of splits that should be carried out. (The default one is -1 indicating to spli until the end of the string).
.join(iterable)	Concatenates elements of a list containing only strings to create a single string. The method applies to a separator (ex string_seprator='/') while the list is the argument between brackets. Ex: string_separator.join(list)

LIST FUNCTIONS AND METHODS

Some built-in functions can be used on lists:

Function	Description
len()	Returns the length of the list (the number of elements)
min()	Returns the element starting with the smallest value. (If the list contains mixed elements, the function displays an error). It distinguishes between lower and uppercases.
max()	Returns the element starting with the largest value. (If the list contains mixed elements, the function displays an error). It distinguishes between lower and uppercases.
list()	Converts an object in a list.
sorted()	Returns a sorted list in ascending order. (If the list contains mixed elements, the function displays an error)

sum()	Returns the sum of the elements of the list (only when they are all numbers).

Methods can also be used with lists (some don't work with mixed values).

Method	Description
.append(element)	Adds new elements to the list. It can be used in loops instructions (remembering to create an empty list in which will be added values).
.insert(index,element)	Inserts the element in the desired position specified in index. The precedent element on the list won't be deleted, only moved.
.remove(element)	Finds and deletes the element the first time it appears.
.pop([index])	Finds the element corresponding to the index and deletes it. If there is no index specified, the function will eliminate the last element.
.extend(list2)	Adds to the indicated list2 to the initial list
.index(element)	Finds the element and displays the corresponding index (if it appears more than once it will return the first index).
.sort()	Sorts in ascending order the elements.
.reverse()	Inverts the element of the list. **N.B.** If we wish to keep the original list, we must copy it and then modify it. (Same foes for sort).
.clear()	Removes all elements in a list
.count(element)	Counts how many times an element appears in the list
.copy()	Returns a copy of the list

Ex:

```
>>> numbers = [ ]
>>> for i in range (0,100,15)
        Numbers.append(i)
```

In the end the list will contain the values: [0,15,30,45,60,75,90]

CROSSING OF STRINGS AND LISTS

It is possible to cross sequences, to find determined values or elements, in three different ways:

- Indications or slicing, to select single elements of the string or list.
- Loops, to set instructions to carry out on each element.
- Some methods and functions are necessary to set up loops or instructions.

We can use the for loop:

```
word = 'bank'
for letter in word:
        print(letter.upper())
```

We could also use a while loop:

```
word = 'bank'
index = 0
while index< len(word):
        print(letter[word])
        index = index +1
```

In both cases the result would be:

```
B
A
N
K
```

If we wanted to substitute or insert values into a list (ex: compute the square of only even numbers):

```
numbers = [1, 12, 11, 55, 10, 6]
for letter in range (len(numbers)):
        if numbers [n] % 2 == 0:
                numbers [n] = number [n]**2
```

[1, 12, 55, 10, 6]

[1, 144, 11, 55, 100, 36]

The first list is the initial list, the second is the final one where only even numbers have been squared.

TWO-DIMENSIONAL LISTS

It is possible that some elements in a list are lists; in this case we talk about two-dimensional lists. It is important to keep in mind that each element/list belonging to the main list has a given index, but in every sub-list, the contained elements also have an index. To refer to single elements of the sub-list we write: main_list[main_index][sublist_index]
To easy the comprehension of this type of lists, rows-columns representations are often used.
For example, if we had this list: client = [['Red',40,'OK'],['Green',25,'NO']] we would represent it this way:

	[0]	[1]	[2]
[0]	'Red'	40	'OK'
[1]	'Green'	25	'NO'

Di indexing is done in this way:
main_list[row_index][column_index]
Ex: Green index is client[1][0]

TUPLES AND DICTIONARIES

TUPLES

A **tuple** is a collection of elements written with **round brackets** and separated by a **comma** (in reality it could also work without brackets, but when using some Python functions, it could no longer understand where the tuple starts and ends). A tuple can be made of all types of elements: numbers, strings, other tuples, etc.
Ex: numbers = (10, 20, 30)
At first sight the tuple could be mistaken for a list (they are indeed quite similar, both being function or operation arguments and responding to indexing and slicing) but there is an important difference: tuples are **unchangeable**, which makes them faster and safer.

It is possible to convert a list in tuple (using the "tuple" command) and a tuple in list (using the "list") .
Ex: list_numbers = list(tuple_numbers)

numbers2= (6, 24, 41, 90, 18, 37, 57, 5, 71)

Command	Meaning of the slicing	Result
numbers2[4]	Selects the element of the tuple of index 4	18
numbersi2[-6]	Selects the 6th element starting from the right.	90
numbers2[2:5]	Selects the elements starting with index 2 (included) until index 5 (excluded)	(41, 90, 18)
numbers2[:3]	Selects the elements from the first to the one with index 3 (excluded)	(6, 24, 41)
numbers2[:7:2]	Selects the elements from the first to the one with index 7 two by two.	(6, 41, 18, 57)
numbers2[-5:-2]	Selects the elements from the one indexed -5 to the one indexed -2 (included)	(18, 37, 57)

TUPLES OPERATORS, FUNCTONS AND METHODS

numbers = (1, 2, 3)
city=(Milan, London, Paris)

Operator	Operation
+	Joins two tuples in a new tuple.
*	Creates more copies of a tuple and joins them.
in	Returns True if the element is in the tuple
not in	Opposite of in

Ex: new_tuple=numbers+city = (1, 2, 3, Milan, London, Paris)
Ex: new_tuple=city*2=(Milan, London, Paris, Milan, London, Paris)

Here are the **built-in** functions that can be used with tuples:

Funzione	Descrizione
len()	Returns the numbers of elements in a tuple
tuple()	Converts an object (list) in a tuple
max()	Returns the element starting with the largest value. It distinguishes between lower and uppercases.
min()	Returns the element starting with the smallest value. It distinguishes between lower and uppercases.
sorted()	Returns a sorted tuple in ascending order.
sum()	Returns the sum of the element in the tuple.

We move on to methods.

Method	Description
.index(element)	Returns the index of the searched element, if there is more than one, it takes the first.
.count(element)	Returns the number of times an element appears in a tuple.

CROSSING OF TUPLES

The elements of a tuples can be selected in sequences, we use:

- Slicing and indexing if we want to select one single element
- Loops if we want to apply the instruction to each element of the tuple
- Some functions and methods to set up combinations of loops and instructions

Ex:

```
even_numbers = (2, 6, 88, 74)
new_list=[]
for n in range(len(even_numbers)-1):
        new_list.append(even_numbers[n]/2)
```

```
>>> new_list
[1.0, 3.0, 44.0]
```

DICTIONARIES

A dictionary is an object containing a collection of data. The **elements** of each dictionary are built out of:

- A **key**, that is unchangeable and univocal
- A **value**, that can be both changeable or unchangeable

Here is how we write a dictionary:
Ex:

```
dictionary_name = {'key1' : value1, 'key2': value2}
```

To add a new element to the dictionary (or to modify or overwrite an existent one) we do:

```
dictionary_name[keyname]=value_name
```

To access the stored values:

```
dictionary_name[key_name]
```

To delete a dictionary or an element of the dictionary we use the **del** command:

```
del dictionary_name
del dictionary_name[key_name]
```

To access to more values at the same time, we must use loops.

DICTIONARIES OPERATORS, FUNCTIONS AND METHODS

With dictionaries, given the elements structure made of **key** and **value**, it is not possible to use traditional operators (sum, product, etc.). We use **logic operators**:

Operator	Operation
in	Returns True if a key is found in the dictionary, False if not.
not in	Opposite of in
is	Returns True if two dictionaries are the same, False if not
is not	Opposite of is

We can use some **built-in** functions:

Function	Description
len()	Returns the numbers of elements (pairs key-value) in a dictionary
dict()	Creates an empty dictionary, except if an argument of the function is an iterable object.
max()	Returns the largest key. It distinguishes between lower and uppercases.
min()	Returns the smallest key. It distinguishes between lower and uppercases.
sorted()	Returns a list with keys sorted in ascending order (if keys are strings they will be written in alphabetical order).

Here are the dictionary **methods**:

Method	Description
clear()	Deletes every element in the dictionary (doesn't delete the dictionary)
.get(key_name[,default_value])	Returns the value corresponding to the key, if the key is not in the dictionary it will display **None** as predefined value, or, if inserted, the default value (optional)
.pop(key_name[,default_value])	Removes the key and the associated value, diplaying the value. If the key is not in the dictionary, it displays **KeyError** as predefined value, or, if inserted, the default value (optional).
.popitem()	Deletes the last key-value copy added to the dictionary and displays them as tuple.
.update(dictionary2_name)	Adds the keys and their values from a second dictionary in an existent dictionary
.items()	Creates an object **dict_item** that contains a list of tuples (each with a key-value pair)
.keys()	Creates an object **dict_keys** that contains a list with all the dictionary keys.
.values()	Creates an object **dict_values** that contains a list with all the dictionary values.

Clarifications:

- The result of **get** could be obtained using indexing (dictionary_name[key_name]), but in this case if key_name was not part of dictionary_name we would have an error message. Using **get** we get the predefined value **None.**

- Ex: **".pop"**: dictionary_name.pop(key_name[,default_value])
    - If **key_name** is defined in **dictionary_name** we would have as a result the value associated to the key and the elimination of the pair key-value; if **key_name** is not defined in **dictionary_name** we will have **default_value** (if inserted). Finally, if neither **key_name** nor **default_value** have been indicated, we will have a **KeyError.**

- The methods **items**, **key** and **values** returns **iterable objects** called **view objects**; they are not copies of the dictionary but their "**image**" that will update according to changes in the dictionary and are very quick to analyze, making them suitable when working with large quantity of data.

Ex: method.**items** with a for loop:

```
>>> dictionary_name.items()
dict_items([('key1_name',value1_name),(' key2_name',value2_name)])
```

Applying the **for** loop to an hypothetic address book with names and numbers being respectively keys and values we get:

```
for k, v in address.items():
        print("Name:",k, "\Number:", v)
Name:  Marco          Number: 27
Name:  Giovanni       Number: 45
```

ACCESS TO FILES AND ERRORS MANAGEMENT

ACCESS TO FILES

It is possible, within Python to open (or create) text files and modify them.
Two ways to access a file:

- Sequential access (on which we will focus): we access the file and we read it from top to bottom.
- Direct access: we directly access the part that interests us (quicker process)

To open a file we must first of all create a variable to which we associate the so-called "file object", writing variable = open(file,mode)
"file" must appear and has to include the file path and the file name (if the path is omitted, Pyhton will assume its position is the same as the program's). if in the path there are some "\" it is necessary to write it, preceded by a 'r', so that "\" is not considered as an escape character.
variable = open (r'D:\Example\file.txt')
The argument mode, however, is optional and sets the way in which the file should be open (reading only, writing, etc.). The default mode is reading only. Modes in which to view files:

Mode	Description
'r'	Opens the file for reading only. When opened, it starts at position 0 (beginning of first line). If it doesn't exist, an error will display.
'w'	Opens the file for writing. If the file already exists, the already written data will be deleted, else a new file will be created.
'a'	Opens for appending at the end of the file without truncating it. Creates a new file if it does not exist.
'r+'	Opens the file for reading and writing. If the file already exists, data is not deleted, if it doesn't, an error message is displayed.
'w+'	Opens the file for reading and writing. If the file already exists, data is deleted, if it doesn't, a new file is created.
'a+'	Opens the file for appending and reading. If the file already exists, Python positioned at the end of the existing text, else a new file is created.

In order to write and move in a file, it is necessary to use methods, at the end of the changes, we must close the file with the method close ->variable.close().

There exist the so-called attributes that must be written after the name assigned to the file, giving us information on the file:

- variable.name: returns the file path and name
- variable.mode: returns the mode used to open
- variable.closed: returns True or False according to whether the file is open or closed

Contrary to methods, attributes don't have arguments and we don't use brackets.

METHODS TO READ DATA

Methods to read data and position inside the text are a few. The most used are:

Method	Description
.read(size)	Reads the content of the file from the current position until the end. If we insert the size argument (optional), Python reads the set number of bytes. Ex: if size=10 the program reads from the current position to 10 characters after.
.readline(size)	Reads the content of a line of text. Size remains optional.
.readlines(size)	Reads the content of all lines of text until the end and returns a list with the single lines as elements. Size is optional.
.seek(offset [,whence])	Allows to move the pointer to the desired position, specifying from how many bytes we must move. The offset argument indicates the number of bytes, while the whence argument indicates the starting position: 0=beginning of the file, 1=current position, 2=end of file.
.tell	Returns the position in the text in terms of bytes starting from the beginning. (Considers spaces and escape codes)

METHODS TO WRITE ON A FILE

There exist methods to write on an open file in append or write mode.

Method	Description
.writable()	Returns True if it is possible to write on a file, False if not.
.write(string)	Writes in the file thee content insert as string. In the end Python returns the number of characters written and the pointer in the file settles at the end. We must insert in the string argument potential escape codes.
.writelines(lines)	Writes in the file the sequence of lines written in lines. We must also insert escape codes.

ERRORS MANAGEMENT

During the writing and the running of the program, some errors can occur. There are 3 types:

- **Syntax errors:** there is an error in code syntax (writing error or error in the structure of the program) that will block the execution of the program. If we are in the Editor and we want to launch the program, the error Invalid syntax will be displayed, and the error will be highlighted in red. If we are in the shell, a traceback message will indicate the point in which there is an error with the instruction that generated it with a brief description. We have this type of error when for example we forget punctuation or brackets.
- **Runtime errors:** the syntax is correct but there is a mistake in the code. They are caused by operations that cannot be carried out or that are not included in Python. They can be due for example, to variables that haven't been initialized, divisions by 0, use of operators that are not adapted to the data type, etc.
- **Semantic errors:** those errors don't depend on syntax or other, but simply depend on a writing mistake. For example if we wanted to compute the average and we wrote: average = x+y+z/3 we would have a wrong result; we would have to write: average = (x+y+z)/3

TRY … EXCEPT

When we foresee that a given instruction might give an error, it is possible to use the functions try and except to avoid the error message. It is called **exception handling.**
Ex: we want to divide two numbers but the second is zero, or the inserted numbers are strings; to avoid the ZeroDivisionError or ValueError we can write:

```
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!')
```

As show, try and except must be followed by ':' and have to be written with the same indentation. Instructions also need to be indented at the same level.

TYPES OF ERROR

Eccezione	Descrizione
AttributeError	When a method or attribute are assigned to the wrong data/object type.
IndexError	When we use a too large index in a sequence.
NameError	When a value is not found
SyntaxError	When there is a syntax error
TypeError	When an operation or function is applied to the wrong data type
ValueError	When a function or method receives an argument of right type but invalid value. Ex: we try to convert a string in int
ZeroDivisionError	When we try to divide a number by 0

DEBUGGING

Debugging is the removal of errors, of all type, and the optimization of the program.
It is an experimental process that occurs both during and after the writing of the code. It is possible to use several instruments for the debugging: we have the pdb module that is in the standard Python library, that basically analizes the code, or we can install external packs, both free and not.

PYTHON LIBRAIES

Arguments treated in this chapter may be considered more difficult for some. It is advised to try the arguments progressively as you read.

MODULE: DEFINITION

A fundamental part of coding is to reuse the code in order to increase time efficiency when programming, to achieve that we use **modules**.
Modules are **script** files (.py) that contain pieces of code with specific functions, they are organized by **use purpose** and **homogenous content**.
In modules we find: **functions**, **object classes** (specific data types), **methods** (specific functions made to deal with attributes, ex ".sort()"), **docstrings**, **comments** and **example data** (for complex functions).
There are three types of modules:

- Already installed in Python, in the **standard library**;
- Created by the user, that must be searched for every time you open Python;
- Developed by other programmers, downloadable on designated websites (GitHub and PyPI) to save time.

If we had to build particularly complex program it would be smart give a **hierarchical approach of type top-down to the code**, we will use, beside modules, library containing akin modules for **use purpose** and **homogeneous content.** If necessary, we could also use larger libraries with sub-libraries.
Once we upload the library, we must tell Python which function (in which module, in which sub-library of which library) to use. To avoid confusion, we resort to the writing convention **dot notation**, in which we always start from the most general to end with the most particular. For example, a program to upload an image would be:
>>> manageImage.reading.typeJPEG.openFile("C:\\path\\MyPhoto.jpg")

In this case the logic path is:
Library.sublibrary.module.function(argument)

To import modules and libraries in Python we use the **import** function, with which we can important several modules and libraries. Ex:
>>> import math, random

N.B. modules and libraries are imported from the RAM, so every time the program is restarted (in our case the Shell), we will need to reimport the modules and libraries needed.

As we might only need a single functions from a library or a module we can use the function **from** that imports only given functions and not the whole library (or module). Ex:
>>> from math import sqtr

N.B. in this case the math library wouldn't be recognized by the program thus would not be usable.

There are some functions useful to understand which modules are present in Python while working. This functions must be used in the following way since they are "support functions":

- **help:** discovers what modules are installed on the hard disk (regardless of their origins) and is written
    - >>> help('modules')

- **dir:** discovers which modules are active in the working session and is written:
    - dir()

The Python build-in module (and its internal functions) will appear in the form **'__builtins__'**, while imported modules will appear in the form **'maths'.**
It is possible to use the dir() function by inserting a module or library as argument:
>>> dir(math)
['sqrt', 'ceil', etc.]

THE PYTHON STANDARD LIBRARY

The standard Python library contains over 180 modules, more or less complex.

THE MATH MODULE

It allows to work on real numbers and most used functions:

Function	Description	Example
math.ceil(x)	Returns the integer rounded by excess	4.7 → 5
math.floor(x)	Returns the integer rounded by default	4.7 → 4
math.sqrt(x)	Returns the square root	6.25 → 2.5
math.pi	Expresses Pi with approximation according to the PC power	3.14...
math.exp(x)	Returns e to a given power	
math.factorial	Returns the factorial of positive integer	4! → 4*3*2*1

math.gcd(a, b)	Returns the greatest common divisor of a and b	Tra 36 e 84 → 12
math.hypot(x, y)	Applies Pythagoras's theorem to compute the	rad(4*2+3*2) → 5
math.isfinite(x)	Returns True if x is different from 0/n and 0/0, else False	
math.isnan(x)	Returns True if x is equal to 0/0, else False	
math.log(x [, baseA])	Returns the logarithm in base A. If baseA is omitted, it computes the logarithm in base 10	
math.logq (x)	Computes logarithm in base q of x	
math.pow(x, y)	Elevates x to the power y	3**4 → 81

THE RANDOM MODULE

Allows to use random functions for probability calculations.

Function	Description
random.random()	Returns a decimak number between 0 (included) and 1 (excluded)
random.choice(sequence)	Returns a random element between those indicated in the sequence.
random.randrange(stop) random.randrange(start, stop[,step])	Returns an integer number chosen between **start** (included, and if omitted = 0) and **stop** going **step by step** (if omitted =1); the functioning is the same as the range function.
random.randint(min, max)	Returns a random integer between numbers between **min** and **max** (both included).
random.seed(a = None)	Initilaizes the generator of random numbers with the integer **a** given; if omitted or **None**, it uses the value of the internal clock of the computer at the execution moment. We use it if we always want to obtain the same sequence of random values.
random.shuffle(x)	Shuffles randomly the order of the elements of sequence x
random.sample(population, k)	Returns a list of **k** elements from the **population** (without putting-back)
random.sample(population, weights = None, k = 1)	Extracts a list of **k** elements from the **population** sequence doing samples (putting-back). It is possible to make the elements equiprobable by adding **weights**, a value equal to the length of population that contains the different probabilities of the elements.
random.gauss(mu, sigma)	Generates a random number from a normal random variable with given **mu** and **sigma**.

THE TURTLE MODULE

Allows a very simple graphic use.

Turtle: methods	Description
bob = turtle.Turtle()	Build a new turtle object called **bob.**
bob.forward(distance)	The **bob** turtle moves forward of pixel **distance**
bob.backward(distance)	The **bob** turtle moves backward of pixel **distance**
bob.goto(x, y)	The **bob** turtle goes to position x=abscise and y=ordinate
bob.left(degrees)	The **bob** turtle rotates anti-clockwise of given degrees.
bob.right(degrees)	The **bob** turtle rotates clockwise of given degrees.
bob.setheading(degrees)	The **bob** turtle orients itself with respect to the original position by degrees indicated
bob.penup()	Lift the **bob** turtle's pen so that it stops writing.
bob.pendown()	Lowers the **bob** turtle's pen so that it writes.
bob.begin_fill()	The **bob** turtle starts filling the figures drawn.
bob.end_fill()	The **bob** turtle stops filling the figures drawn.

bob.pencolor(color)	Sets up the color of **bob** turtle's pen.
bob.width(dimension)	Sets up the width of **bob** turtle's pen.
bob.fillcolor(color)	Sets up the color **bob** turtle will use to fill the figures drawn.
bob.shape('turtle')	Changes the aspect of **bob** turtle (ex; with 'turtle' it will look like a turtle)
bob.position()	Indicates the current position of **bob** with coordinates given in a tuple (x,y)
bob.heading()	Indicates the degree positioning of **bob.**
canvas = turtle.Screen()	Builds a new object named **canvas.**
canvas.bgcolor(color)	Sets the color of the background window.
canvas.addshape("ninja.gif")	Adds to the the window an image as shape of the pointer.

THE DATETIME MODULE

Gives special data connected to dates.

Main data types	Description
x = datetime.date(year, month, day)	Creates a value of type date with attributes: **year, month, day**.
y = datetime.time(hour, minute, second)	Creates a value of type **time** with attributes: **hour, minute, second**.
z = datetime.timedelta([days[, seconds[, minutes, [, hours[, weeks]]]]])	Creates a value **duration** obtained by the difference between two values of type **date** or **time.**
x.weekday()	Returns the day of the week as an integer number **0-6**, where Monday is 0 and Sunday is 6.
x.isoformat()	Returns a string representing a date in format: YYYY – MM -DD.

CLASSES, ATTRIBUTES AND METHODS

OBJECT-ORIENTED PROGRAMMING

Python is a multi-paradigm programming language, meaning that it uses different programming styles in which:

- Procedural approach, that decomposes the program in coding blocks (sub-programs or sub-routines), identified by a name and enclosed in delimitations, in such a way to create a sort of pyramidal hierarchy.
- Functional approach, in which the program takes the shape of a sequence of functions to carry out in a concatenate manner.
- Programming approach, object-oriented (OOP), which is the way Python itself was developed.

OBJECT MADE LANGUAGE

In Python, everything is made of objects and, as in real life, every object can be used in one or more ways, according to the type of object we use.
There are two types of actions that we can carry out; the ones entailing the modification of characteristics of attributes of the object, and the ones that use the object in its interaction with other objects or with the user.
In Python, objects are characterized by attributes, and by methods relative to objects. Each code can be reused, it is then necessary to make it the least complicated possible according to the Don't Repeat Yourself (DRY) logic.

The principles at the basis of OPP are three:

- **Encapsulation:** characteristic that makes invisible to other objects the characteristics of a class. It presents many advantages, such as the simplicity to share with other programmers, an easier and quicker testing of the functioning, the security of the code, an increase in productivity, etc.
- **Inheritance:** construction mechanism of classes that allows to build new ones from already existent ones, inheriting attributes and methods (we talk about child classes and parent classes).
- **Polymorphism:** mechanism according to which we can give the same name to methods that act the same way on different objects.

CLASSES AND INSTANCES

The class is a family, a set of objects of a certain type that can be already present in Python or in external libraries, or can be created by us.
An instance is the single object present in the class.
To better understand, let's assume that our class is the automobile class, instances would be the automobile designs.
Each object must have particular attributes and methods. To go back to our example, attributes could be the power of the engine, the type of gearbox, etc., while methods could be the way the car starts, switches off, change gear, etc.
The set of methods of a class is called interface.

Inside Python, there are types of elementary data (strings, integers, float, bool) and more complex data structures (tuples, lists, dictionaries) that are called literals.
Contrary to real objects, literals are created with a dedicated syntax, easier and provided with identification elements, that allow Python to spot the literals (for example strings are identified by (' ') or (" "), integer numbers by figures between 0 and 9 and so on).

If, when we talked about variables and types of data, we saw that it is possible to change data type by using functions (str, int, float), in reality what happens is that the conversion doesn't modify the original object but it creates a new one using the first, reassigning the desired name and type. In Python indeed, the type is strictly linked to the object and cannot be modified.

As we have already mentioned, there exist classes already present in the Python standard library and in external libraries. There exist some precise code-naming rules that allow the identification when using or creating a class; one of those rules is starting the name of the class with an uppercase letter. If we use the type function, we can verify at which object class the element belongs to, while to see the content of the module we must use the dir() function.
Ex: >>> dir(turtle) will show the entire content.

If we wanted to view only attributes and methods, we should use dir() with the name of the instance or the class.
Ex: >>> dir(turtle.turtle()).

Finally, to also have an explanation, we can use both the help and the tip function that is viewed when writing the name of an object followed by a full stop or when we open brackets after a method.

CUSTOMIZED CLASSES

To create objects, it is necessary to define and create a class.
In reality, no class is created from scratch, each on has as a base the class in which the object is re-made.
To create a class, we must first of all define the name. Suppose we want to create a class of objects to play Heads or Tails:

```
class Coin ():          To note that brackets are not compulsory but should be written by convention.
    pass                Should also be written the ":" and the following line should be indented.
```

To enrich the object with attributes, we must build the method indicated by "__init__"; to go back to our example, suppose we want to have as an initial situation before the throw, one of the two sides; __init__ is a function and, as method, it has to have at least one argument that, by convention is called self:

```
class Coin ():
        def _ _ init_ _ (self):
                self.initialSide = 'Heads'
```

The building method doesn't compute anything, it only defines attributes of the future instance, which is why it doesn't end with return.

It is possible to add to the building methof many other attributes, in our case we could add the value of the coin, etc.
Attributes can be variable or constant.
After the creation of attributes, productive functions must be created. In our case the ones that simulate the throw of the coin. The method we will create has to have as argument the same inserted before, self.
In general, the scheme to use for creating a class is the following:

```
class TypeObject ():
        def _ _init_ _(self):
                self.attribute1 = value1
                …
                self.attributeN = valueN

        def method1 (self, mandatoryArguments, optionalArguments = "default"):
                "instruction 1"
                …
                "instruction N"
        …

        def method (self, …)
```

Finally, it is possible to add docstrings inside classes to give explanations and instructions during the use. If we use the help function on the class, we will view these potential docstrings.

INSTANCES

To create a single instance it is necessary to give it a name and assign it to the object of the created class.

```
number1 = Coin ()
```

Afterwards, it is possible to modify the initial values (already set in the attributes); to do this we must call the created object followed by the attribute that we wish to change.

```
number1.currency = "$"
```

If we want to use a method, we must write using this syntax:

```
object.method()
```

It is also possible to delete an object. In this case we use the del keyword:

```
del number1
```

To create a duplicate of an object it is not possible to assign a new name, we should use the copy module. It is a module containing only two functions: copy and deepcopy

```
import copy
object2=copy.copy(object1)
```

CUSTOMIZE INSTANCES ATTRIBUTES

As we mentioned, the argument self is a mandatory parameter that always has to be inserted in the method __init__, however other arguments can be inserted in the method.
They can be optional or mandatory:

```
class TypeObject ():
        def _ _init_ _(self, par1, par2 = 'value')
```

In this case par 1 is mandatory while part 2 is not. When writing attributes it is good to keep in my that there are some specific attributes for each instance and some attributes relative to the whole class. It is usual to write the instance attributes int the __init__ method, and the class attributes outside.

```
class TypeObject ():
        #class attributes
        attribute1 = val1
        attribute2 = val2
        ...
        def _ _init_ _(self, par1, par2 = 'value')
                #instance attributes
                self.attribute3 = val3
                self.attribute4 = val4
```

SPECIAL METHOD __SRT__

This method allows to build a representation of an object in a string format. This method is recalled each time Python needs to return an object in string format (print(), str(), format()).
As every class comes down from the base object class, if the __str__ is not defined in a class, Python will get it in object.

In the string constituted by __str__, the programmer can insert all sort of information useful for the user of the object.
Another characteristic is having as a unique argument self and to always need to finish with a return instruction.

BEST PRACTICE WHEN CREATING INSTANCES

To keep other programmers from editing the content of the code, it is possible to make certain attributes constant. To do so, we use the get functionality. If we want to make the attribute modifiable, we must use the set functionality.

```
def getCurrency(self):
        return self.currency
```

```
def setInitialize(self, newSeed):
        self.initialize = newSeed
```

INHERITANCE

Inheritance is based on a hierarchical concept, that in Python is pretty easy to create and use.
It is sufficient to indicate in the subclass (child) the name of the base class (parent).

```
class SuperClass():
    pass
class SubClass(SuperClass):
    pass
```

METHODS OVERRIDING

We call overriding the mechanism according to which Python, according to the name of the method, prioritize the execution, in the subclass, of the methods of the subclass itself. This is because we suppose that the level of specialization of the lowest levels is greater.

```
class Employee ():
        hours = 8
        hWage = 15
        def wage (self):
                print ("The wage is €"+self.hours*self.wage*20)

class Parttime (Employee):
        def wage (self):
```

```
        print("As you are a part time worker, your wage is €" +self.hours*self.wage*20*0,5)
```

If we looked for the part-time wage, using the Parttime class we would obtain the execution of the second module called wage.

METHODS OVERLOADING

We talk about overloading of methods when operators take different functionalities according to the type of data used. This happens through the use of particular methods (for example __add__).
It is a characteristic that we have already encountered with operators + and * that work differently according to whether they operate on stings or number (decimal or integer).

POLYMORPHISM

To finish with, polymorphism is the characteristic that allows the program to decide in which way to use a given function or method, according to the type of data it faces (one same function can indeed act in different ways on different types of data).