Ministry of Higher Education and Scientific Research

Ibn Khaldoun University Tiaret

Faculty of Applied Sciences

Department of Science and technology

Course materials

Field : Science and Technology

Subject *: Python Programming (Computer Science 04)*

Intended for : Second Year Engineering (ST)

Manuscript prepared according to the officially approved and confirmed program of Ministry of higher education and scientific research

Established by:

Dr. *BENSATTALAH Aissa(MCA)*

Experts :

Mr. MAASAKRI Mostafa (MCA)

Mr.MAATOUG Abdelfettah (MCA)

Year 2025/2026

In an increasingly digitally shaped world, programming has become a new form of literacy. Among the languages that open the doors to this universe, Python stands apart with a singular philosophy: one of clarity, simplicity, and elegance.

Regardless of the software used (Matlab, Octave, Maple, or other software,
Or programming Language Python, Java, C, C++, Delphi, Fortran, the basic principle of programming remains the same. Programming is based on the concept of algorithms, meaning that you must master algorithms to program.
Algorithms are a way of thinking that transforms a problem into sequential instructions, conditional instructions, and/or repetitive instructions.

An algorithm consists of a set of well-organized instructions designed to solve a given problem e.g An algorithm A step-by-step procedure to solve a problem. Then a program is a sequence of instructions written by a programmer in a given language to solve a specific problem.
Generally, there are three types of instructions:

- Sequential instructions;
- Alternative or conditional instructions;
- Repetitive instructions.

Programming is used in almost every scientific field, and Python is not simply a technical tool; it is a language designed for the human brain. Its clean and intuitive syntax often closely resembles natural language. This handbook on Python programming is prepared for second-year of Science and Technology students, as well as for various specializations such as civil engineering, electronics, automation, and mechanical engineering. The handbook developed in accordance with the officially approved curriculum for the second year of engineering studies.

Table of contents

Chapter 1: Installing and Using Python

The Python programming language has been developed since 1989 by Guido van Rossum. Python is a cross-platform, portable, dynamic, extensible, and free language.

Python is a popular programming language for several reasons:

- **Simple and readable**: Its clear syntax makes it accessible to beginners
- **Versatile**: Can be used for web development, data science, artificial intelligence, automation, and more
- **Active community**: Numerous resources and libraries available
- **Free and open source**

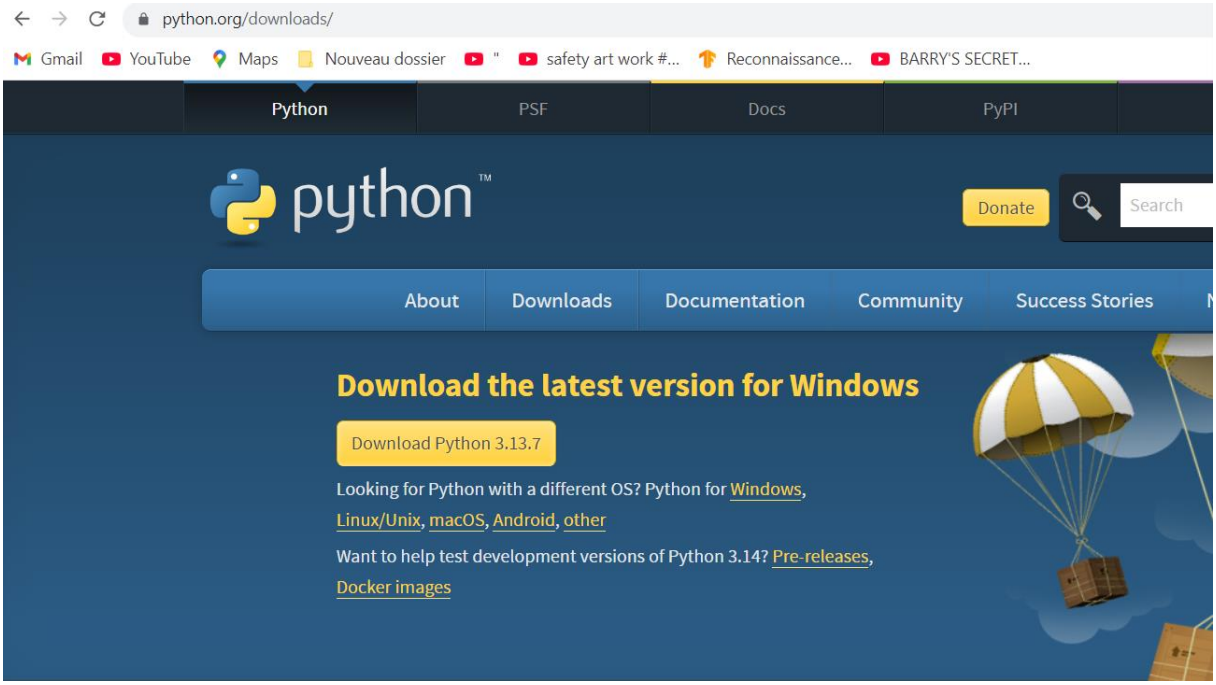
Chapter 1 presents the different techniques for installing Python or techniques for using it without installing it.

Installing Python On Windows

1. Go to the official website: https://www.python.org/downloads/

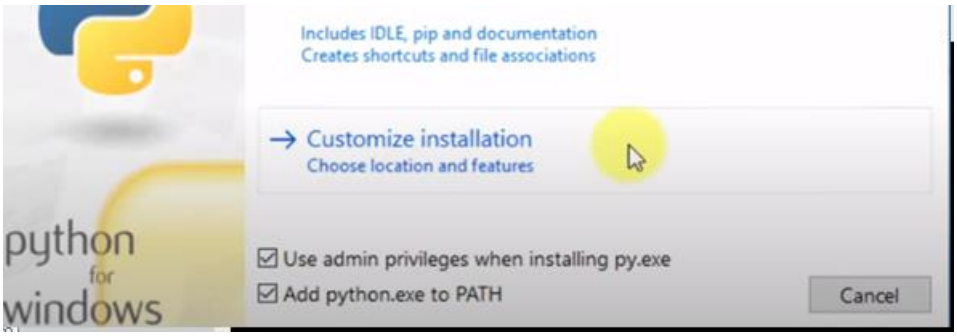
2. Choose the version compatible with your operating system (Windows, macOS, Linux).

3. Download the installation file and run it.



Figue I-1: download python

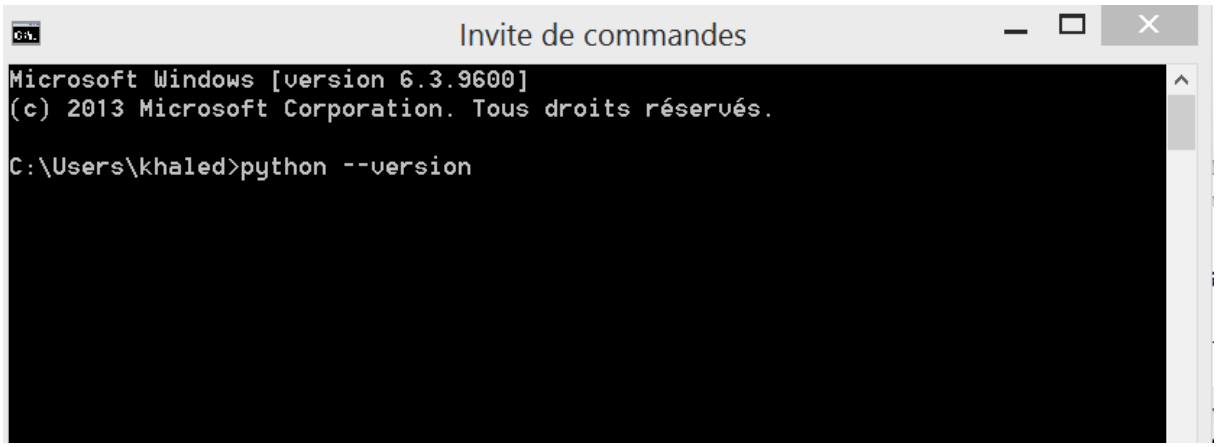
During installation, check both boxes to ensure Python installs correctly.



Figue I-2: installing python

Verification:

1- Open Command Prompt (cmd)
2- Type python --version
3- You should see the installed version

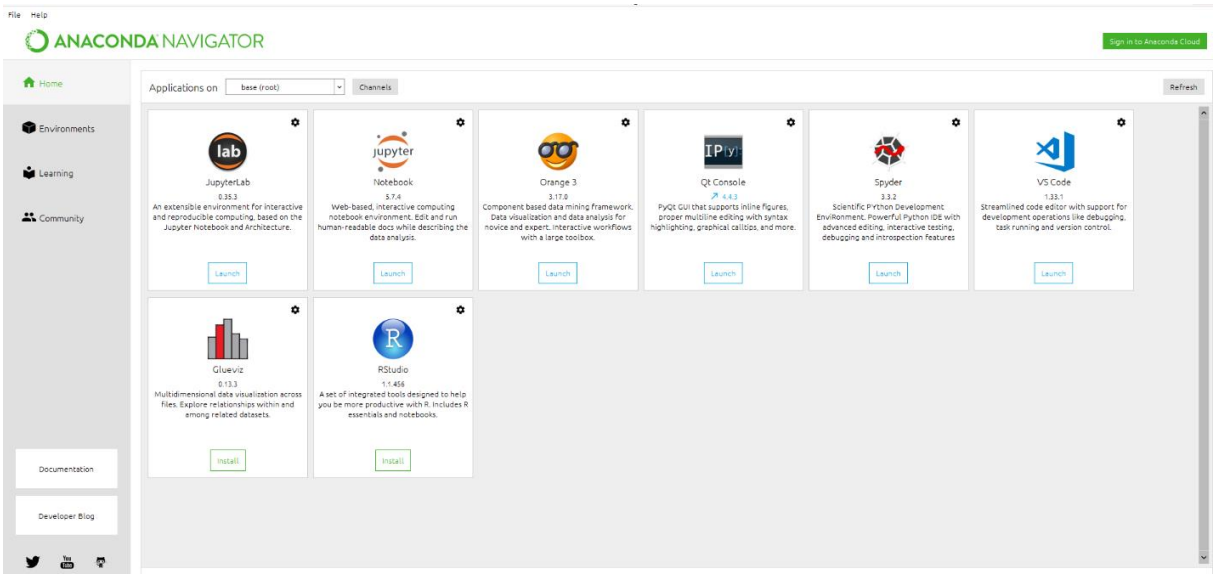


Figue I-3: **Verification**:

• If the Python version is displayed, the installation is successful and you can start using Python.

Installing Python via Anaconda

Anaconda is a free and open-source distribution[2] of the Python and R programming languages applied to the development of applications dedicated to data science and machine learning (large-scale data processing, predictive analytics, scientific computing). Anaconda includes a complete set of essential packages for data analysis.

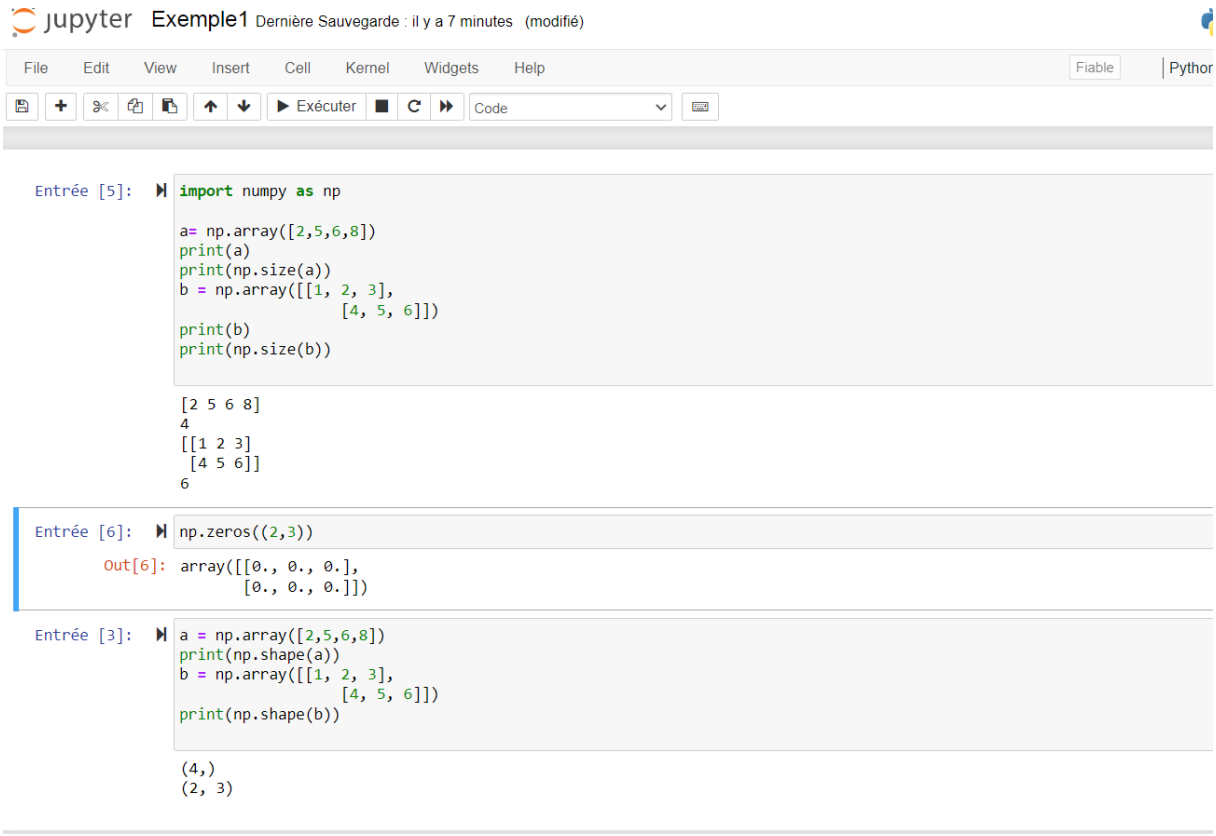


Figue I-4: Anaconda

The Anaconda Browser is a graphical user interface (GUI) included in the Anaconda distribution that allows users to launch applications (see figure) and manage conda libraries, environments, and channels without using any command lines. The Browser can also access libraries located on the Anaconda Cloud or in a local Anaconda Repository to install, run, and update them in an environment. It is available for Windows, macOS, and Linux.

The following applications are available by default in the browser: JupyterLab, Jupyter Notebook, and Spyder.

Jupyter selection for programming with Python (figure)



Figue I-5: Jupyter

To install a package using pip, you can use the command:

```
pip install <package_name>
```

Numpy package

The term NumPy is short for "Numerical Python," it is a popular Python library primarily used for mathematical and scientific calculations.

Matplotlib package

Maptolib is an open-source Python library for creating scientific and statistical data visualizations

Recommended Tools

Code Editors

VS Code: Lightweight, Python extensions available

PyCharm: Specialized for Python (free Community edition)

Sublime Text: Fast and customizable

Development Environments

Jupyter Notebook: Ideal for data exploration

Google Colab: Jupyter in the browser, no installation required (figure)

Using online-python https://www.online-python.com/ (figure)

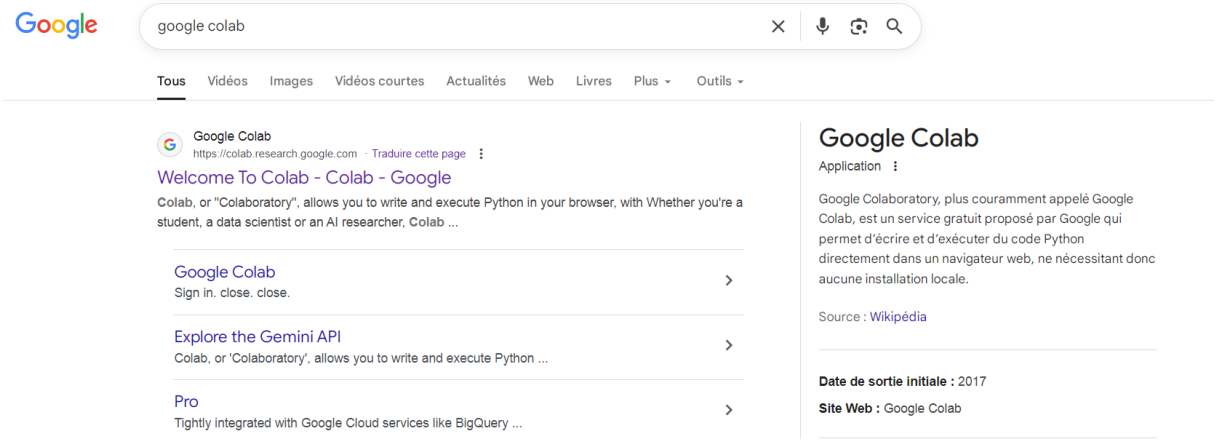
Using Google Colab

Go to the Website: Open your browser and navigate to colab.research.google.com

Sign In: If you aren't already, sign in with your Google account (like your Gmail).

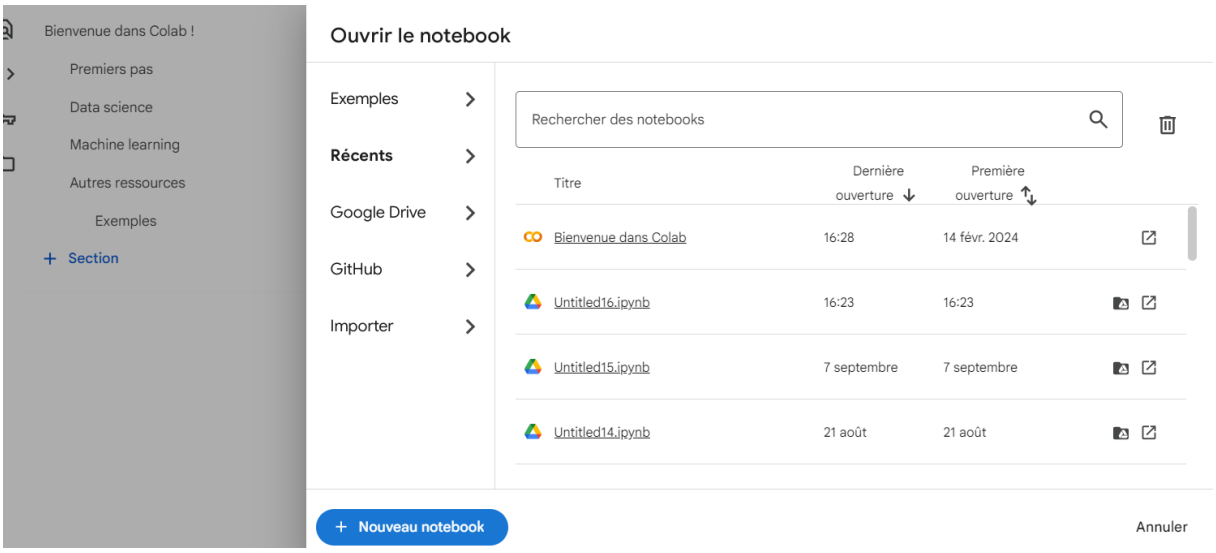
Create a New Notebook: On the welcome screen, click on **"New Notebook"**. This will open a new coding environment based on Jupyter Notebook (figure).

Start Coding: You can immediately start writing and running Python code in the cells. The file is automatically saved to a folder named "Colab Notebooks" in your Google Drive (figure)



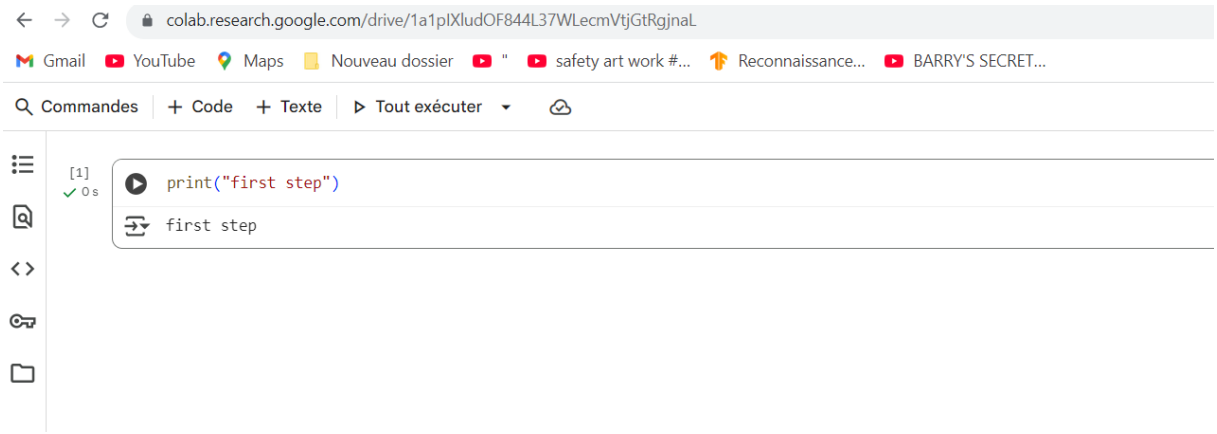
Figue I-6: colab.research.google.com

Simply connect via your email



Figue I-7: Jupyter Notebook

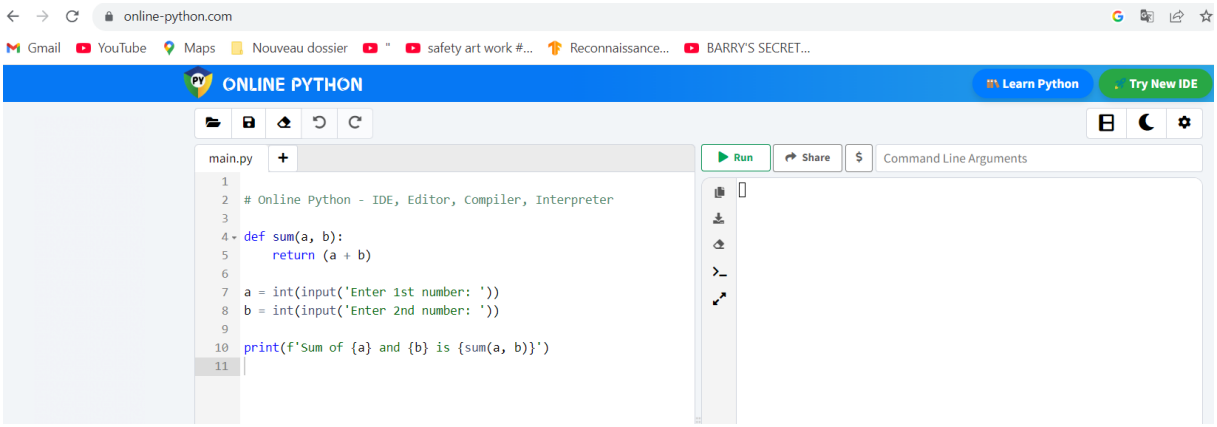
start writing and running Python code in the cells(see figure)



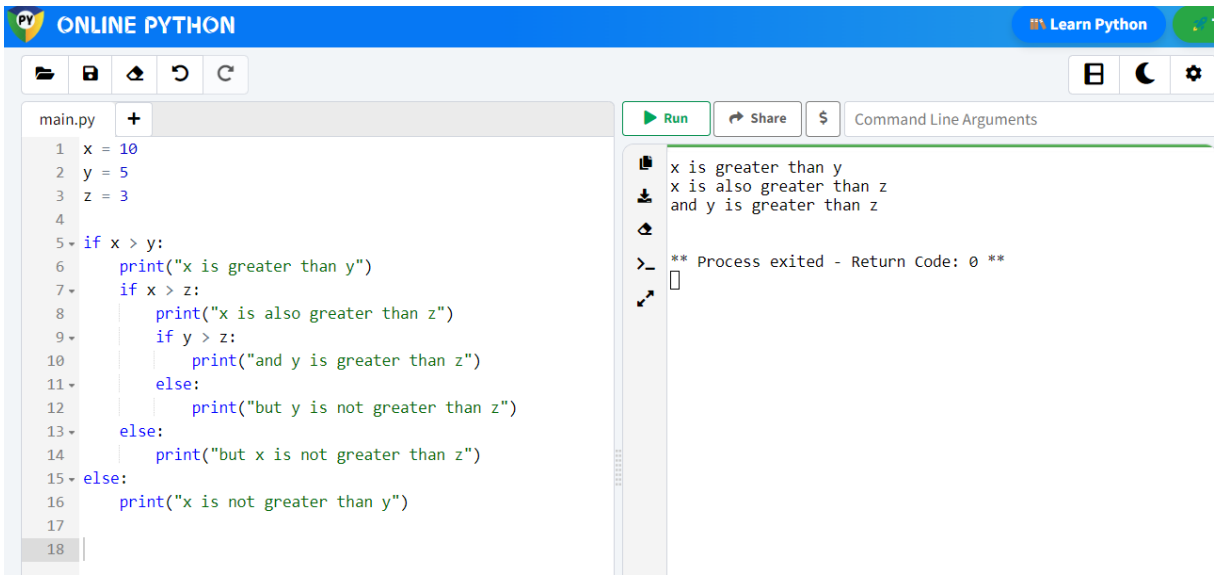
Figue I-8: writing and running Python code

Using online-python.

Online-Python.com is a free browser-based tool that lets you write, run, and share Python code without any installation. It's a straightforward choice for quick coding practice, debugging, and learning the basics.



Figue I-9: online-python.



Figue I-10: Example online-python.

Chapter 2: Basic Concepts

Chapter 2 introduces the basics of Python, including interactive and script modes for executing code. It covers operators, data types, type conversion, using functions from the math module, formatted output, the print function, the input function, variable naming conventions, and comments.

2-A. interactive mode and script mode

There are two ways to use Python (Interactive mode and script mode) for execute code, each with distinct purposes and workflows

2-A-1. Interactive Mode

In interactive mode, also called console mode, the interpreter allows to enter instructions one by one; simply press the ENTER key for the interpreter to execute them.

Figure 2 below shows the Hello World program in interactive mode. The lines that begin with are the command prompt that allows you to enter an instruction.

```
>>> print('Hello World!')
Hello World!
```

```
>>> x = 5
>>> y = 10
>>> x + y
15
```

Python's interactive mode, also called REPL, allows you to execute commands one by one, ideal for quick calculations or tests, while script mode executes a .py file containing a sequence of instructions, perfect for more complex programs.

- **Best for**:
  - Testing small code snippets
  - Learning and experimentation
  - Quick calculations
  - Debugging parts of code

2-A-2. Script Mode

Writing code in a .py file and executing it as a program in ordre **to use** first create a file (e.g., myscript.py) and run python myscript.py

- **Best for** :
  - Building full applications
  - Reusable code
  - Complex programs
  - Automation tasks
  - Production code

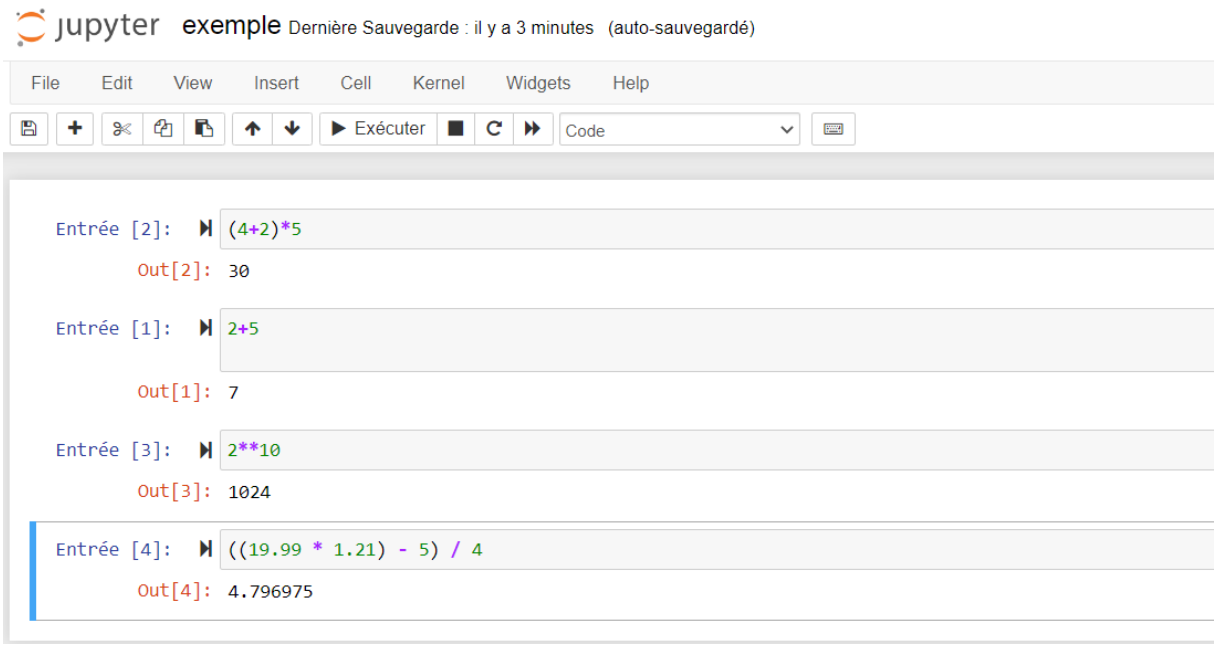
2-A-3. Key Differences

Feature	**Interactive Mode**	**Script Mode**
Execution	Line-by-line	Entire file at once
Persistence	Commands not saved	Code saved in file
Use Case	Testing/learning	Development/deployment
Output	Immediate	After full execution
File Extension	Not needed	.py required
Feature	Interactive Mode	Script Mode

Use both modes together—writing functions in scripts and testing them interactively before integrating into larger programs!

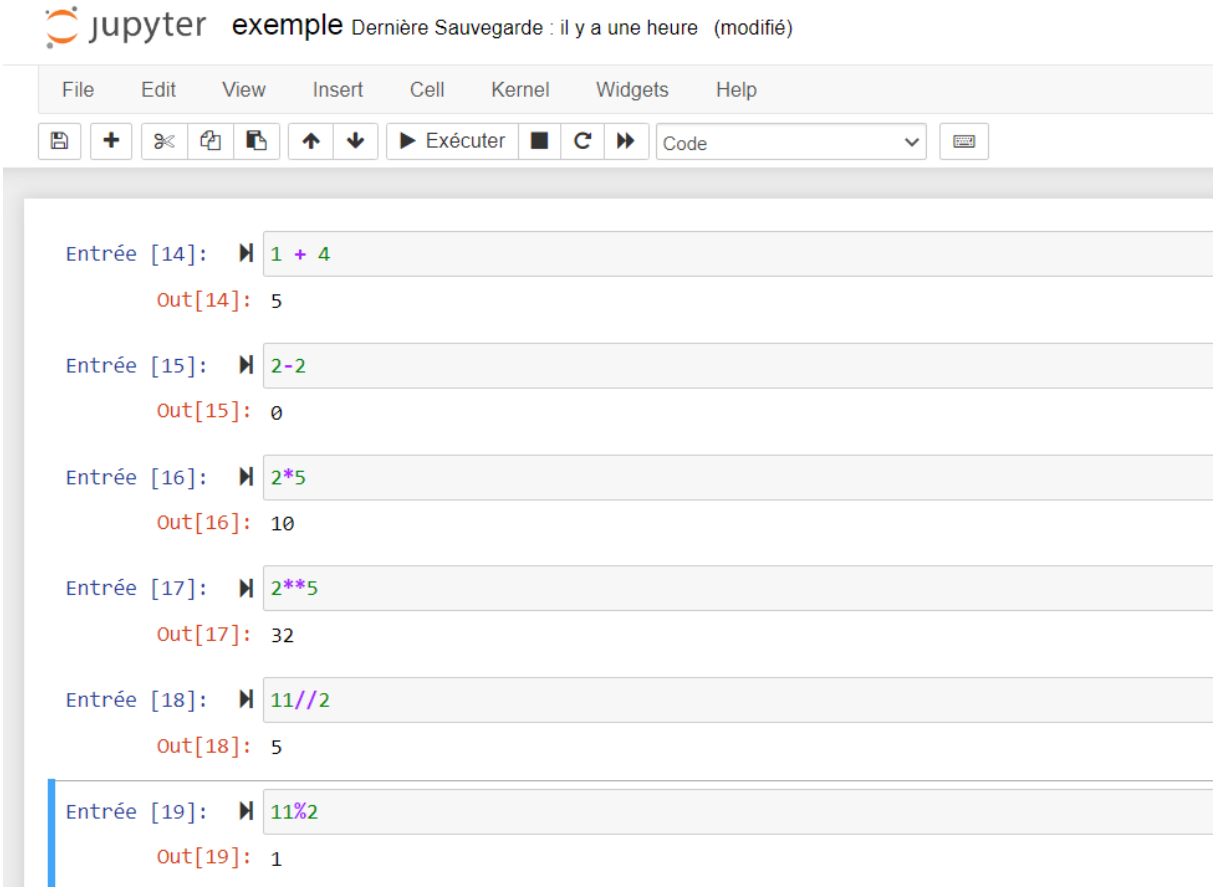
2-A-1. Python Calculator

The Python interpreter can be used in interactive mode like a calculator. For example, a sequence of instructions encoded in interactive mode, along with the resulting output. The mathematical expression; it will be evaluated and its result displayed as an intermediate output.



2-A-2. The use of operators: +, -, *, /, //, %, et **,

Python offers several arithmetic operators: addition (+), subtraction (-), multiplication (*), division (/), exponentiation (**), integer division (//), and remainder (%).



2-A-3.c Precedence

Each arithmetic operator has a precedence that defines the order in which operations are executed. In order, the exponentiation operator is executed first, followed by the *, /, // and % operators, and finally the + and - operators.

1 + 4 * 5=1 + (4 * 5)=21

2*5**3=2*(5**3)=250

When an expression contains multiple operations of the same precedence, they are evaluated from left to right.

2-5-7=(2-5)-7=-10

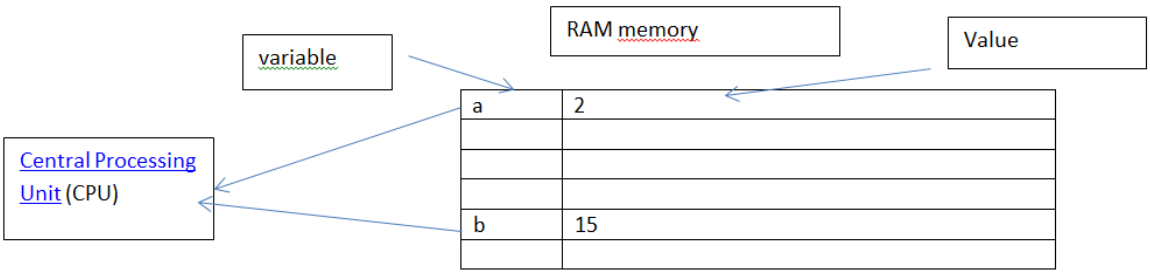
1-3-6-4-9=((((1-3)-6)-4)-9)=-21

To avoid problems, always use parentheses to make the evaluation order of arithmetic expressions explicit.

2-B. Variable and data type

In programming, we need to define variables to store results, and also to retrieve them. In real problems, we use multiple variables.

A variable in a programming language serves as a named storage location for a value. It allows programmers to store and manipulate data within a program.



>>>a=2

>>> b=2

Allocate memory space for variable a and assign the value 2;

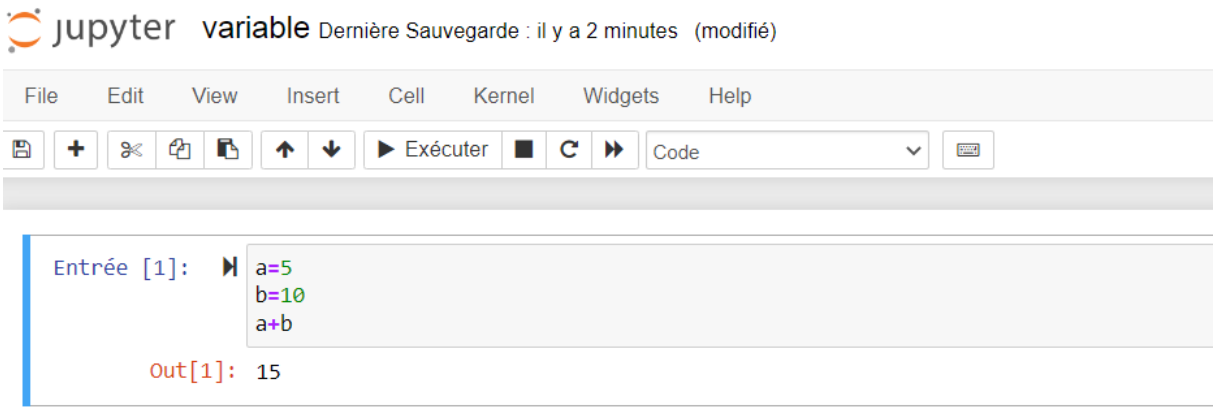
Allocate memory space in RAM for variable b and assign the value 15.

RAM, or Random Access Memory, is your computer's short-term, high-speed memory used to temporarily store data and applications that the Central Processing Unit (CPU) needs to access quickly to perform tasks,

A varaible is a memory space used to store a value. It has a name, a value and a data type

Name	value	type
a	2	integer
b	15	integer

To create a variable, we use the simple structure: ***variable = expression***



Both 'a' and 'b' are variables. For example, the variable 'a' could contain the number 5 and the variable 'b' could contain the number 10. During execution of the program, the statement "a + b" is replaced by the Actual Values "5 + 10" and the result becomes 15.

Example of calculate the force using the force law F=m*a, with m=10kg and a=23m/s2, in this example there are three variables F,m and a.

A 25Ω heating resistor carries a direct current of 9.5 amperes. What is the voltage supplying this heater, given that Ohm's law is: U=R×I, where: U in volts (V) ; I in amperes (A) ; R in ohms (Ω).

In this example there are three variables U,I and R.

A person of mass 60 kg climbs onto a chair 60 centimeters high. How much work does this person do?. Know that the work law is W = -mgd, where g = 9.81 represents the gravitational constant, 60 is the person's mass in kilograms, and 0.5 is the height in meters.

in this example there are four variables W,m,g and d.

2-B-1. Variable initialization, Variable modification, Compound assignment

In Python, variable initialization and modification are performed using the assignment operator `=`.

Initialization is the assignment of a value to a variable at the time of its declaration, i.e., the first time it is used.

Modification (or assignment) is the assignment of a new value to an already existing variable during the course of the program.

Compound assignment uses operators such as `+=`, `-=`, `*=`, etc., to combine an operation and an assignment; for example, `x += 3` is equivalent to `x = x + 3`. Compound assignment is an operation that combines an arithmetic or logical operation with an assignment in a single statement.

`=` represents assignment in Python, not equality as in mathematics.

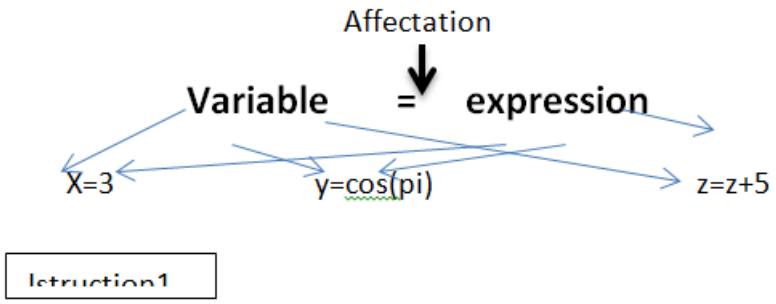


Figure:Instruction in python

Initializing and Modifying a Variable

• **Initialization:** This is the first time a value is assigned to a variable. Python then creates the variable in memory.

Example:

x=3. # Initialize the variable x with the value 3

• **Modification:** To change the value of an existing variable, use the assignment operator

Example:

x = 31 # Modification of variable x, new value 31

x = 60 # Modification of variable x, new value 60

x = 42 # Modification of variable x, new value 42

Overwrite the old value in memory and keep the last modification.

- **Compound Assignment**

Compound assignment is a shorter syntax for performing an operation on the current value of a variable, then reassigning the result to that same variable.

• Syntax: variable operator = expression

• Meaning: The expression variable op = expression is a contraction of

variable = variable op expression.

Examples:

• x += 5 is equivalent to x = x + 5.

• y -= 2 is equivalent to y = y - 2.

• z *= 3 is equivalent to z = z * 3.

• a /= 4 is equivalent to a = a / 4.

2-B-2. Data Types

Variables hold different types of data, such as whole numbers (int), text (String), numbers with decimal points (double, float), and true/false values (boolean, bool).

Integers : value is represented by int class. It contains positive or negative whole numbers (without fractions or decimals).

Float : value is represented by float class. It is a real number with a floating-point representation. It is specified by a decimal point. Optionally, character e or E followed by a positive or negative integer may be appended to specify scientific notation.

Complex Numbers : It is represented by a complex class. It is specified as (real part) + (imaginary part)j. For example - 2+3j

```
x = 50  # int
x = 60.5  # float
```

```
a = 5
print(type(a))

b = 5.0
print(type(b))

c = 2 + 4j
print(type(c))
```

String Data Type : Strings are an example of a sequential data type. A sequence is an ordered collection of items . Sequences allow storing of an organized multiple values.

Python Strings are arrays of bytes representing Unicode characters.

Strings in Python can be created using single quotes, double quotes or even triple quotes. We can access individual characters of a String using index. Indexes are basically *measures of position*. All elements within a variable start at index 0.

w	e	l	c	o	m	e	t	o	u

```
Text=’Welcome to  university of Tiaret’

print(text)

# check data type

print(type(text))

# access string with index

print(text[1])

print(text[2])

print(text[10])
```

Boolean : Data Type in Python is one of the two built-in values, True or False. Python can evaluate any expression, and get one of two answers, True or False. When you compare two values, the expression is evaluated and Python returns the Boolean answer:

```
print(10 > 9)
print(10 == 9)
print(10 < 9)
```

When you run a condition in an if statement, Python returns True or False:

Example

Print a message based on whether the condition is True or False:

```
a = 200
b = 33

if b > a:
  print("b is greater than a")
else:
  print("b is not greater than a")
```

2-B-3. Conversion (fonction str)

In Python, the str() function converts any object to a string. It is used to transform data types like integers, floats, or lists into a textual representation, often for ease of display, concatenation, or use in formatting strings. The str() function is Python's **primary tool** for converting objects to strings

```
# Converting integers and floats
my_int = 42
my_float = 3.14

converted_int = str(my_int)    # Result: '42'
converted_float = str(my_float) # Result: '3.14'

print(type(converted_int))     # Output: <class 'str'>
```

without The str() function will show an error because you cannot combine text with a number directly

example

```
age = 25
print(‘mon age est: str(age))
```

The str() function is an essential tool for converting data into a form that can be read, displayed, and manipulated as text, and is essential for combining different data types in Python.

2-C. predefined functions

In Python, predefined functions are functions that are already written and available to the user without needing to be defined. They are built directly into the language and allow you to perform common tasks, such as displaying text with print(), getting the length of a string with len(), or providing input to the user with input().

Some common examples of predefined functions:

print(): Displays information on the screen.

len(): Returns the length (number of elements) of a sequence such as a string or list.

input(): Prompts the user to enter data from the keyboard.

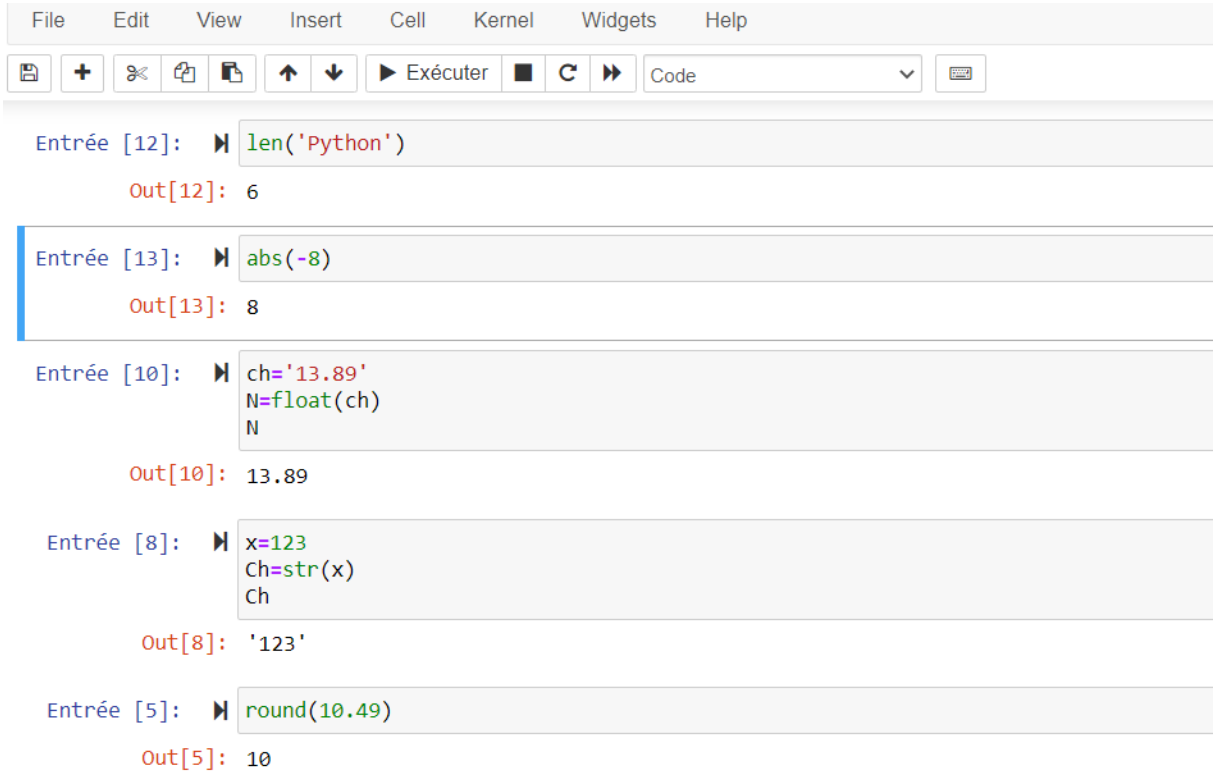
int() and float(): Used to convert values to integers or decimals.

abs(): Returns the absolute value of a number.

round(): Rounds a number.

str(): Retourne la conversion d'un nombre x en une chaîne .

type(): Indicates the data type of a variable.



2-C-1. Use the functions from the math module (abs, max, min, pow, round, sin, sqrt, log, exp, acos, etc)

The math module in Python provides access to common mathematical functions and constants, including trigonometry functions (sin, cos), functions for logarithms and exponentials (log, exp), the square root (sqrt), rounding functions (ceil, floor), functions for complex numbers, and constants like pi (math.pi).

Arithmetic Functions

ceil(x): Rounds x up.

floor(x): Rounds x down.

sqrt(x): Returns the square root of x.

pow(x, y): Calculates x to the power of y.

fmod(x, y): Returns the modulo of x by y (preferred over floating-point operators for floating-point numbers).

fabs(x): Returns the absolute value of x.

factorial(x): Calculates the factorial of x (for positive integers only).

isinf(x): Tests if x is infinite.

isnan(x): Tests if x is "Not a Number" (NaN).

Logarithm and Exponential Functions

log(x[, base]): Calculates the logarithm of x to the specified base (or base e by default).

exp(x): Calculates the exponential e^x.

Trigonometric and Hyperbolic Functions

sin(x): Calculates the sine of x (angle in radians).

cos(x): Calculates the cosine of x (angle in radians).

tan(x): Calculates the tangent of x (angle in radians).

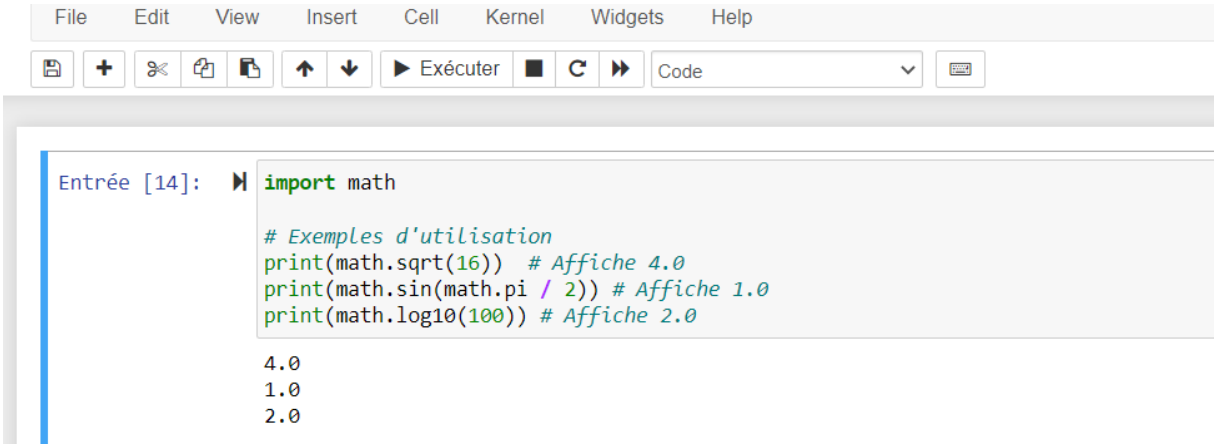
asin(x), acos(x), atan(x): Inverse trigonometric functions.

Constants

math.pi : The constant pi.

math.e: The base of the natural logarithm.

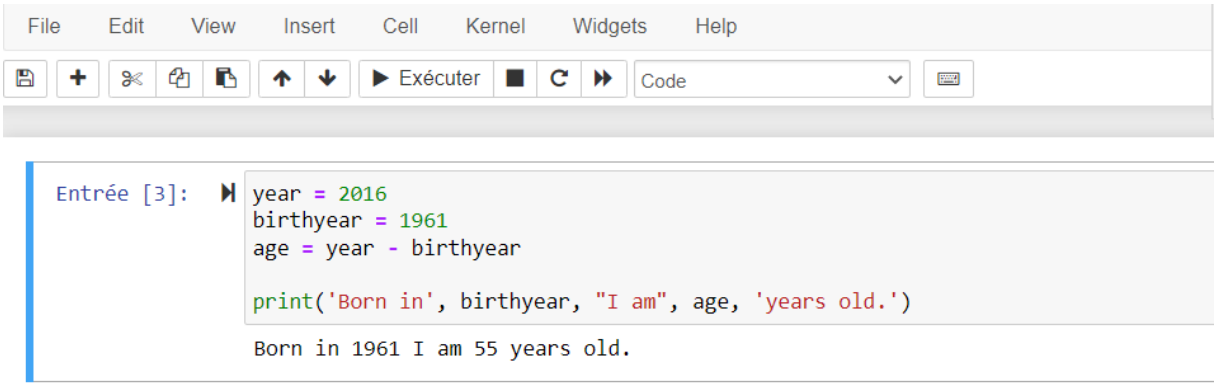
To use these functions, firstly import the module math into your Python script with import math



2-C-2. Function print ()

The print () function using to display data or result of executing a program, regardless of its type.

The following example displays the sentence « Born in 1961, I am 55 years old.» :



2-C-3. Formatted output (using the format function)

Presenting and displaying a program's outputs or execution results in a readable form for a human, or saving the results in a file for future use, is important. Python allows users to present, format, and display data (execution results) according to their needs.

Number Formatting

Number formatting allows you to specify how numbers are displayed and represented in strings. There are several situations, for example, requiring at least three digits to represent a number.

Integers

To display integers, we use the identifier d (for digit). The representation of an integer without modification is written {:d}.

"{:d}".format(54)

It is mandatory to have three digits XXX, even if the number is less than 10.
In this case, you must specify 0n before the d, where n is the minimum number of digits.
Therefore, if you format the number 8 with {:03d}, the number will be formatted as 008.

```
for i in range(0, 120, 8):
  print("{:03d}".format(i))
```

If you do not want zeros but spaces instead, simply omit the digit 0 in the formatting.

```
for i in range(0, 120, 8):
  print("{:3d}".format(i))
```

Numbers with decimal points

Formatting is used to limit the writing of digits after the decimal point.

For numbers with commas, the identifier **f** (for floating point) is used.

```
"{:2.1f}".format(85.456984155)
```

The {:2.1f} format reads like this.

At least two digits are displayed before the decimal point.

Formatted string (*f-strings*)

Formatted strings (also called f-strings) allow you to enclose the value of Python expressions in strings by enclosing them in curly braces {} after prefixing the string with an f or F .

placeholders {}, a placeholder can contain variables, operations, functions, and modifiers to format the value.

The following example rounds pi to three decimal places:

```
>>> import math
>>> print(f'The value of pi is approximately {math.pi:.3f}.')
```

The value of pi is approximately 3.142.

Example

Add a placeholder for the price variable:

```
price = 59
txt = f"The price is {price} DA"
print(txt)
```

output : The price is 59 DA

Example

Perform a math operation in the placeholder, and return the result:

```
txt = f"The price is {20 * 59} DA "
print(txt)
```

The price is 1180 DA

A modifier is included by adding a colon : followed by a legal formatting type, like .2f which means fixed point number with 2 decimals:

Example

Display the price with 2 decimals:

```
price = 59
txt = f"The price is {price:.2f} DA "
print(txt)
```

The price is 59.00 DA

String format()

format() method in Python is a tool used to create formatted strings. By embedding variables or values into placeholders .

The format() method also uses curly brackets as placeholders {}, but the syntax is slightly different:

String Format() Syntax

string.format(value1, value2, ...)

Parameter: values (such as integers, strings, or variables) to be inserted into the placeholders in the string.

a = "Mohamed" # name

b = 22 # age

msg = "My name is {0} and I am {1} years old.".format(a,b)

print(msg)

Output

My name is Mohamed and I am 22 years old.

format(a, b) method replaces {0} with the first argument (a = "Mohamed") and {1} with the second argument (b = 22).

Example

Add a placeholder where you want to display the price:

```
price = 49
txt = "The price is {} DA"
print(txt.format(price))
```

The price is 49 DA

Multiple Values

If you want to use more values, just add more values to the format() method:

And add more placeholders:

Example

```
quantity = 3
itemno = 567
price = 49
myorder = "I want {} pieces of item number {} for {:.2f} DA."
print(myorder.format(quantity, itemno, price))
```

I want 3 pieces of item number 567 for 49.00 DA.

Index Numbers

You can use index numbers (a number inside the curly brackets {0}) to be sure the values are placed in the correct placeholders:

Example

```
quantity = 3
itemno = 567
price = 49
myorder = "I want {0} pieces of item number {1} for {2:.2f} DA."
print(myorder.format(quantity, itemno, price))
```

```
I want 3 pieces of item number 567 for 49.00 DA.
```

2-C-4. Fonction input

The input() function in Python is used to pause program execution and read a string of text entered by the user via the keyboard.

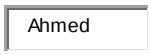
The Input () function allows for user input the data

Example

Ask for user input, the following example asks for your name, and when you enter a name, it gets printed on the screen

```
print("Enter your name:")
name = input()
print(f"Hello {name}")
```

```
Enter your name:
```




```
Hello Ahmed
```

Multiple Inputs

You can add as many inputs as you want, Python will stop executing at each of them, waiting for user input:

Example

```
name = input("Enter your name:")
print(f"Hello {name}")
fav1 = input("What is your favorite animal:")
fav2 = input("What is your favorite color:")
fav3 = input("What is your favorite number:")
print(f"Do you want a {fav2} {fav1} with {fav3} legs?")
```

The input from the user is treated as a string, You can convert the input into a number with the float() function

4. Example with a simple calculation

number1 = int(input("Enter the first number: "))

number2 = int(input("Enter the second number: "))

sum_result = number1 + number2

print(f"The sum is: {sum_result}")

Example

To find the square root, the input has to be converted into a number:

```
x = input("Enter a number:")

#find the square root of the number:
y = math.sqrt(float(x))

print(f"The square root of {x} is {y}")
```

2-C-5. Importing functions

To use functions that are part of a module, you must first import them. This is done using the `from/import` statement, specifying the module name after `from` and the list of functions to import after `import`.

```
from math import sqrt, sin,pi

print('Square root of  16 =', sqrt(16))
print('Sine of pi = ', sin(pi))
print('pi=',pi)
```

```
Square root of  16 = 4.0
Sine of pi =  1.2246467991473532e-16
pi= 3.141592653589793
```

2-D. Source code

In Python, there are best practices for writing correct and high-quality source code. You can't just write code haphazardly; you must adhere to the variable naming conventions.

2-D-1. Variable naming rules

In Python, variable names must begin with a letter or an underscore (_), contain only alphanumeric characters and underscores (A-z, 0-9, _).

Fundamental Rules:

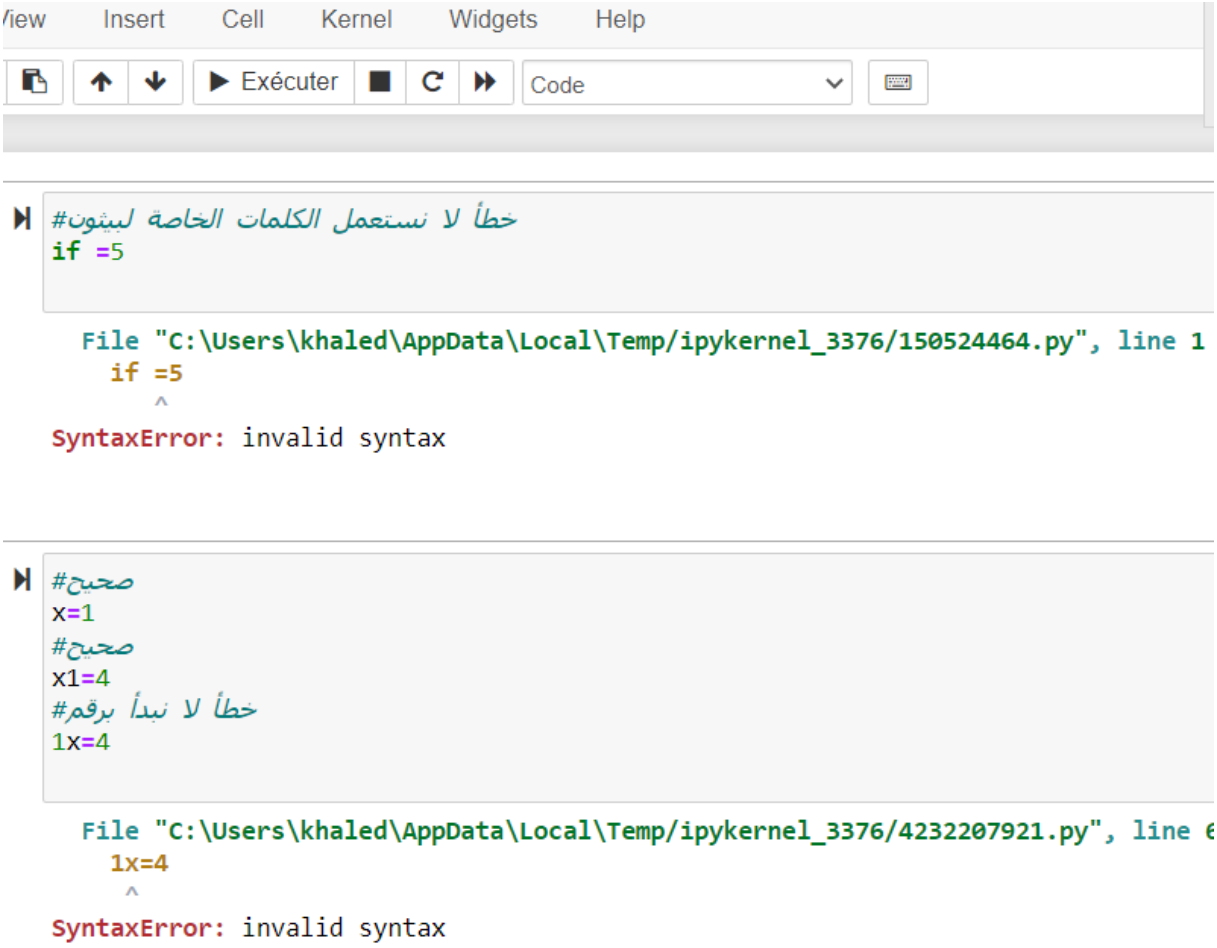
Allowed characters: Letters (a-z, A-Z), digits (0-9), and the underscore character _.

Prohibited: No spaces or accents.

Beginning: Must not begin with a digit.

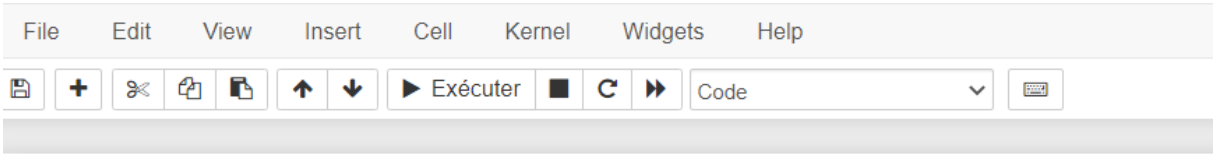
Case sensitivity: variable, Variable, and VARIABLE are three distinct variables.

Keywords: Do not use words reserved by the language (print, if, for, def, etc.).



2-D-2. comment

In Python, a comment is a piece of code ignored by the interpreter that serves to explain the code. The hash symbol # is used for single-line comments, and triple quotes (''' or """) are used for multi-line comments, which are often used as documentation strings (docstrings).



Entrée [6]:

```
# Programme de calcul des racines d'une équation du second degré
# de la forme ax^2 + bx + c
# TP2 : département ST

#importation du module math
from math import sqrt

# Coefficients a,b et c
#   x^2+3x+2=0
a = 1
b = 3
c = 2

# Calcul delta
delta = b ** 2 - 4 * a * c

# Calcul les deux racines
x1 = (-b + sqrt(delta)) / (2 * a)
x2 = (-b - sqrt(delta)) / (2 * a)

print('Les deux racines sont :', x1, 'et', x2)
```

```
Les deux racines sont : -1.0 et -2.0
```

Concepts to recall

variable = expression

Constants math.pi (3,14), math.e (2,71) …

Arithmetic Functions : ceil(x): floor(x): sqrt(x): pow(x, y): fmod(x, y):fabs(x): factorial(x):

Logarithm and Exponential Functions log(x[, base]): exp(x)

Trigonometric and Hyperbolic Functions sin(x): cos(x): tan(x): asin(x), acos(x), atan(x):

Simple Data Types : Integer (int), Floating (float), String (str), Boolean (bool), Complex Numbers (complex), List (list)

Arithmetic Operators : Addition (+), Substraction (-), Multiplication (*), Division (/), Modulus (%), Floor Division (//), Exponentiation (**)

Operator's Precedence : (), *, /, %, **, +, -

Logic Operators : Equal (==), Different (!=), Less than (<), greater than (>), Less than or equal to (<=), Gretaer than or equal to (>=), Conjonction (and), Disjonction (or), Negation (not)

Basic Instruction and Functions : print(), input()

Data Formatting : Integer (%d), Float (%f), String (%s), Define the number of digits after decimal point (%.4f), display integer over 2 charcaters (%2d)

Converting Data Types : int(x), float(x), str(x), complex(x), etc

Assignment Management : (a=2, a+=3, a=b=1, a,b=2,3, a,b=b,a)

Exercises

Exercise 1

Insert the parentheses and give the result

1-2*5**3

What placeholders are used when dealing with f-strings?

() ;{} , ||, ‘ ‘ or []

Exercise 2

Write a program that takes two numbers as input and prints their sum.

Solution:

```
#python Program

num1 = int(input("Enter the first number: "))

num2 = int(input("Enter the second number: "))

sum = num1 + num2

print("The sum is:", sum)
```

Exercise 2 Write a program that calculates the area of a rectangle given its length and width.

Solution:

```
#python Program

length = float(input("Enter the length of the rectangle: "))

width = float(input("Enter the width of the rectangle: "))

area = length * width

print("The area of the rectangle is:", area)
```

Exercise 3 : Write a program that takes two numbers as input and prints the largest number.

Solution:

```
#python Program

num1 = float(input("Enter the first number: "))

num2 = float(input("Enter the second number: "))

largest = max(num1, num2)
```

print("The largest number is:", largest)

Exercise 4 (Logic)

Calculate the logical expressions :

- a=0, b=1,
- c=a==1 , d=~b, a<b, a~=b
- e=~(c|d), e1=(c & d)

Exercise 5 (Character String)

Texte= ‘’The objective of the subject is to teach the student programming using easy-to-access software (such as: Matlab, Scilab, Mapple, etc.). This subject will subsequently serve as a tool for carrying out practical work on numerical methods in S4..’’

- Calculate the text length.

- Select only the "programming using" portion of the text.

- Convert the text to uppercase.

```
# Original text
text = "The objective of the subject is to teach the student programming
using easy-to-access software (such as: Matlab, Scilab, Mapple, etc.). This
subject will subsequently serve as a tool for carrying out practical work
on numerical methods in S4."


# 1. Calculate the length of the uppercase text
text_length = len(text)
print(f"Length of the uppercase text: {text_length}")


# 2. Select only a part of the text (e.g., programming using)

part_of_text = original_text[52:70]
print(f"Original part of text: {part_of_text}")


# 3. Convert the selected text to uppercase
uppercase_text = text.upper()
print(f"Uppercase text: {uppercase_text}")
```

Consider the following program, execute and explain

```
text = "’The objective of the subject is to teach the student programming using easy-to-access software "
vowels = "aeiouAEIOU"
   count = 0
   for char in text:
      if char in vowels:
         count += 1
print(count)
```

Exercise 6 (complex number)

Calculate the argument and modulus of the complex number 3+4i

Calculate (3+4i)*(6+2i)

Calculate (3+4i)+(6+2i)

Exercise 7

Execute the code with explanation

```
import cmath

z = 2 + 3j

print(cmath.sin(z))

polar_coords = cmath.polar(z)

print(polar_coords)
```

Chapter 3 : Conditional Structures

Regardless of the software used—Matlab, Octave, Python, Maple, or any other—the basic principle of programming remains the same. Programming is based on the concept of algorithms; that is, mastering algorithms is necessary to program.

Algorithms are a way of thinking that transforms a mathematical problem into sequential instructions, conditional instructions, and/or repetitive instructions.

An algorithm consists of a set of well-organized instructions designed to solve a given problem. Generally, there are three types of instructions:

- Sequential Structures;
- Alternative or conditional Structures;
- Repetitive Structures.

This chapter presents the Conditional Structures :

- ✓ **IF Statement**
- ✓ **IF-ELSE Statement**
- ✓ **Multiple Conditions Statement**

Instructions

Instructions are the actions performed or executed by an algorithm or program.

Sequential Instruction (Structures)

Sequential instructions are the organization of instructions one after another, for example:

Sequential Instructions (or instruction , block , Structures 1)

```
a=5 ;
b=15 ;
X=a*2 ;
Y=b^2 ;
Z=x+y ;
```

Sequential Instructions (or instruction , block , Structures 2)

```
a=5 ;
b=15 ;
```

```
c=20 ;
X=a*2 +c;
Y=b^2 ;
Z=x+y ;
H=x+y+z ;
```

Conditional Structures

Figure II-1 illustrates the principle of executing an alternative or conditional instruction. A conditional instruction is an instruction that, before execution, must verify a condition; if this condition is true, then instruction block 1 is executed; otherwise, if the condition is false, then instruction block 2 is executed.

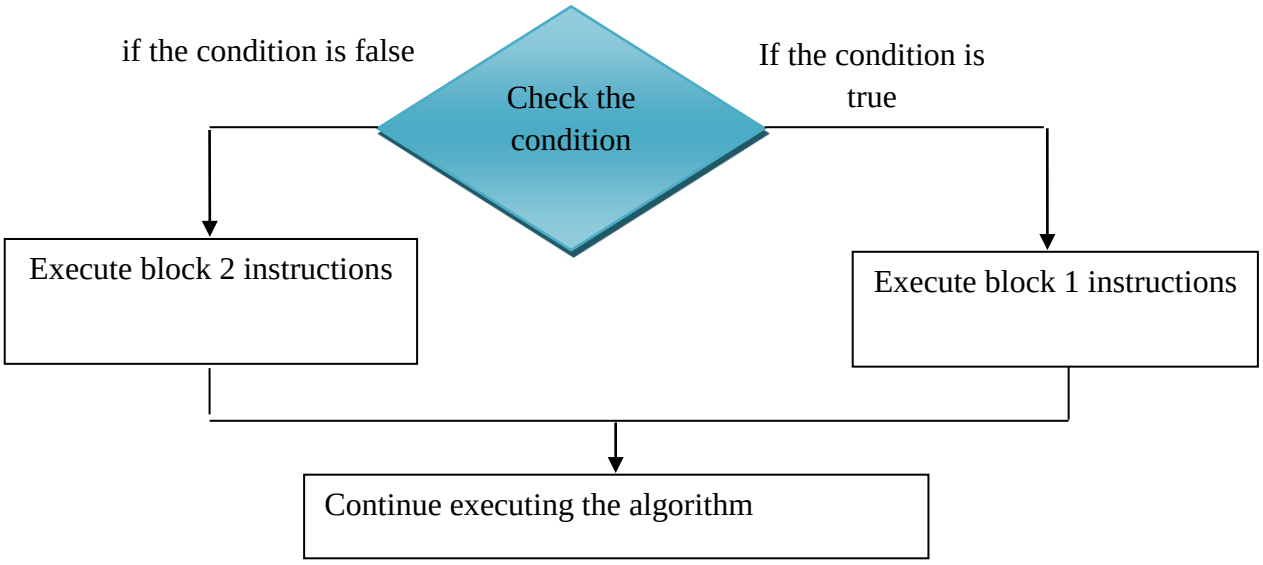


Figure II 1: Flowchart of the conditional instruction

In Python, a conditional structure is written using the keyword if :

Example:

If cos(pi) == 0 then X = 2; Y = 5; else X = 8; Y = 9; end

First, you need to be able to identify the condition; in this example, the condition is cos(pi) == 0, and this condition is false because cos(pi) == -1, so we will execute the following block: else X = 8; y = 9

The syntax of the **if** must be:

1. End the first line of **if** with **":".**

2. An **indentation** corresponds to four spaces or a "tab" keystroke that make up the body of the **if**.

Simple conditional structure (one choice)

```
if condition:
    code
    ...

continuation of the code
...
```

Conditional structure with two choices

the keyword else allows a condition to be executed if the previous conditions are not met.

```
if condition:
    code
    ...

else :
    code

continuation of the code
...
```

Conditional Structure with Multiple Choices

`elif` (for "else if") allows you to add additional conditions to a conditional structure. You can add as many `elif` statements as needed.

If the condition tested in the `if` statement is False, then the conditions in the `elif` statement(s) will be tested. If all these conditions return False, then the code in the `else` statement will be executed.

```
if condition:
    code
    ...

elif :
    code

elif :
    code

elif :
    code

else :
    code

continuation of the code
...
```

Nested Conditional Structure

A nested conditional structure in Python involves placing an if (or if-else) statement inside another if or else structure, thus creating multiple levels of conditions—that is, nesting one conditional structure within another, then within another, and yet another, and so on. A nested conditional structure is used to create more complex, multi-step decision logic. It is also used to test several successive conditions to determine the path to follow in your code.

```
if condition1:
    code
    ...

else :
     if condition2 :
         code
     else :
          code
continuation of the code
...
```

Example

This example shows a nested structure to check if a number is positive, and then if it is greater than 10

```
nombre = float(input("Enter a number: "))

if nombre >= 0:  # Condition
    print("The number is positive or zero.")
    if nombre > 10:  # Condition
        print("It is also superior to 10.")
    else:
        print("It is not superior to 10.")
else:
    print("The number is negative. ")
```

Example : Multiple Levels of Nesting

```
x = 10
y = 5
z = 3

if x > y:
   print("x is greater than y")
   if x > z:
      print("x is also greater than z")
      if y > z:
         print("and y is greater than z")
      else:
         print("but y is not greater than z")
   else:
      print("but x is not greater than z")
else:
   print("x is not greater than y")
```

Exercises

Exercise 1

Write a program that checks if a number is positive, negative, or zero.

Exercise 2

Write a program that display « «AA » if a numbre is 7 ; «BB » if a numbre is 80 ; «CC » if a numbre is 100 ; «DD» if a numbre is 250; «ZZAA » if a number Otherwise:

Exercise 3

Write a Python program that asks the user for a child's age. Then, it informs them of their age category:

"Poussin" (6-7 years old)

"Pupille" (8-9 years old)

"Minime" (10-11 years old)

"Cadet" (12 years and older)

Exercise 4

Write a program that solves the the root of second-degre equation:

The program ask the user enter a,b and c

The program display the solution with message

$$ax^2+bx+c=0$$

Exercise 5

Create a program that determines if a given year is a leap year (divisible by 4, but not by 100 unless also divisible by 400)

Chapter 4: Repetitive Instruction

A repetitive instruction is a block of instructions that repeats. Loops in Python (while, for) allow you to repeatedly execute blocks of code. `while` executes code as long as a condition is true. `for` iterates through iterables (lists, range()) for a defined number of iterations. `break` stops the loop, while `continue` jumps to the next iteration.

In algorithms or programming, there are two types of repetitive instructions:

- ✓ For loop instruction (for);
- ✓ While loop instruction (while);

Repetitive instructions can be used to manipulate lists, tups, dictionaries, strings, etc. (see in chapiter 6 and 6)

In this chapter presents the for loop, Nested loops, the while loop and the break and continue keywords

For loop

The purpose of the "For" instruction is to repeat a sequence of instructions a certain number of well-defined times.

```
for initial value  to  final value do
Instruction block;
End For
```

The syntax of the for loop in python is :

```
for incremented variable in increment range :
    Instruction block;
```

the counter variable, i, to then start the loop.

The syntax of the for **loop** must be:

1. End the first line of for with **":".**

2. An **indentation** corresponds to four spaces or a "tab" keystroke that make up the body of the **loop**.

Example

We want an algorithm that calculates the sum of the first n natural numbers. For example, for n=4, we want to obtain the value of 0+1+2+3=60+1+2+3=6. The pseudocode representation of the program for this example:

The pseudocode

1 S ← 0

2 for i←0 to n-1

3 S ← S+i

4 write S

The range() Function

The range() function in Python generates a sequence of numbers. The range() function is often used with the for loop. It allows to define the number of times of the for loop will be repeated.

range(stop) : Generates a sequence of integers from 0 to stop - 1.

range(start, stop): Generates a sequence of integers from start to stop - 1.

range(start, stop, step): Generates a sequence of integers from start to end - 1, advancing by steps at each iteration.

```
# Creating a list of numbers
numbers = list(range(1, 11))
# [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

# Sum of numbers from 1 to 100
total = sum(range(1, 101))
# 5050
```

Using the range() with the for loop for iteration

```
for i in range(start, stop, step):
    # Code block to execute
```

Exemple de range(stop)	**Exemple de range(start, stop)**	Exemple de **range(start, stop, step)**
for i in range(5): print(i)	for i in range(2, 7): print(i)	for i in range(0, 10, 2): print(i)
Generate numbers from 0 to 4 Output: 0, 1, 2, 3, 4	Generate numbers from 2 to 6 Output: 2, 3, 4, 5, 6	Generate even numbers from 0 to 8 Output: 0, 2, 4, 6, 8

In the exemple flow The range(7) c'est-à-dire **range(stop)** demande à Python d'exécuter l'instruction print("Université") 7 fois.

```
for i in range(7) :
  print("University of Tiaret")
```

University of Tiaret

University of Tiaret

University of Tiaret

University of Tiaret

University of Tiaret

University of Tiaret

University of Tiaret

Lists are the simplest iterable objects in Python.

```
for i in [“Alger”, “Tiaret”,”Oran”] :
  print(i)
```

The list [“Alger”, “Tiaret”,”Oran”] contains three objects of type “string” (a string of characters in Python, i.e., words). The letter i successively takes on all the values of the list.

The result of executing this loop will therefore be:

Alger

Tiaret

Oran

Iterating through a list by index

```
fruits = ['apple', 'banana', 'cherry']
for i in range(len(fruits)):
   print(fruits[i])
```

Optimizing Loops with Range

Minimize instructions within the loop: To improve performance, keep the loop body as small as possible. Use appropriate steps: Adjust the step in Range to skip unnecessary iterations and optimize execution time.

Generating Complex Sequences

Use Range to generate complex numerical sequences that follow a specific pattern.

```
# Generating a sequence of even numbers between 0 and 20
even_numbers = [i for i in range(0, 21, 2)]
print(even_numbers)

# Generating a geometric sequence (powers of 2)
powers_of_2 = [2**i for i in range(10)]  # 2^0, 2^1, ..., 2^9
print(powers_of_2)
```

Nested Loops

A nested loop is a loop inside a loop.

The "inner loop" will be executed one time for each iteration of the "outer loop":

```
adj = ["red", "big", "tasty"]
fruits = ["apple", "banana", "cherry"]

for x in adj:
  for y in fruits:
    print(x, y)
```

and :

```
for x in range(1,3):
  for y in range(1,4):
    print(x, y)
```

The result of executing these loops


```
adj = ["red", "big", "tasty"]
fruits = ["apple", "banana", "cherry"]

for x in adj:
    for y in fruits:
        print(x, y)
```

```
red apple
red banana
red cherry
big apple
big banana
big cherry
tasty apple
tasty banana
tasty cherry
```

```
for x in range(1,3):
    for y in range(1,4):
        print(x, y)
```

```
1 1
1 2
1 3
2 1
2 2
2 3
```

Loop Wihile

The purpose of the "While" instruction is to repeat a sequence of instructions an unknown number of times; that is, execution continues as long as the condition is true. Once the condition becomes false, the instruction stops automatically.

Syntaxe in python :

```
while condition:
    # code block to repeat (الاوامر أو البرنامج المراد تكراره)
```

Warning: Risk of an infinite loop if the condition never becomes false.

Example:

```
counter=0
while counter < 10:
    counter += 1
    print(counter)
```

```
1
2
3
4
5
6
7
8
9
10
```

Keywords: break and continue

The Keywords: break and continue plays an important role in loop management (for and while)

• break: Immediately interrupts the loop (completely exits it).

• continue: Interrupts the current iteration and proceeds directly to the next one.

Example:

```python
matrix = [
    [1, 2, 3, 4, 0],
    [6, 7, 8, 9, 10],
    [11, 12, 13, 0, 15],
    [16, 17, 18, 19, 20],
    [21, 22, 23, 24, 25]
]

line = 0
zero_is_found = False

while line < 5:
    column = 0
    while column < 5:
        if matrix[line][column] == 0:
            zero_is_found = True
            break          # On casse la boucle interne/We break the internal loop(نكسر الحلقة الداخلية)
        else:
            print(matrix[line][column])
            column += 1
    if zero_is_found:
        break              # On casse la boucle externe/We break the external loop/نكسر الحلقة الخارجية
    line += 1

if zero_is_found:
    # N'oubliez pas que les indices commencent à zéro
    print(f"توقف عند ({line}, {column}) : تم العثور على الصفر.")
```

```
1
2
3
4
توقف عند (4 ,0) : تم العثور على الصفر.
```

Exercises

Exercise1 (For loop)

- Calculate the sum of all numbers from 1 to 100

Exercise 2 :Loop Control Statements (break and continue)

- Find the first prime number greater than 100 using a loop and break.

– Print all numbers from 1 to --20, except for multiples of 3, using continue.

Exercise 3 :While loop/do-While loop

- Receive integers from the user until he enter 0, then calculate and print the average of the entered numbers (excluding 0).

Exercise 4

Write a program, which asks user entre text, The program Prints all letters except 'a' and 's'

For example text = “gfazdertsy” **Output (gfzderty)**

For example text = “assma” **Output (m)**

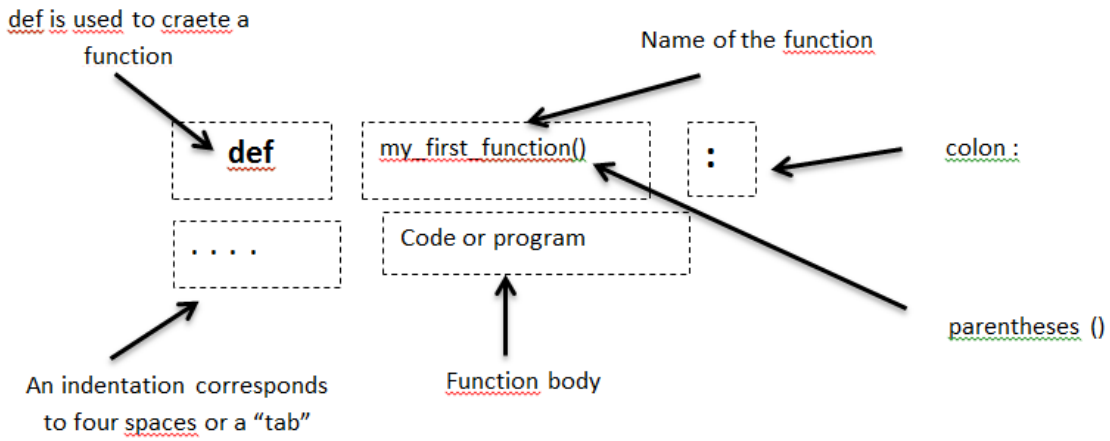
For example text = “samir go to university” **Output (**mir go to university)

Exercise 5

Write a program, which asks for the initial balance K0 and for the interest rate. The program calculate the new capital K1 after one year including the interest. Extend the program with a while-loop, so that the capital Kn after a period of n years can be calculated. Modify it to handle different compounding periods (monthly, quarterly, etc.)

Chapiter 5 : Functions

A function is a block of code which only runs when it is called. You can pass data, known as parameters, into a function. A function can return data as a result. A Functions are defined using the def keyword, followed by the function name, parentheses (), and a colon :



Creating a Function

In Python a function is defined using the def keyword:

Example
`def my_function():` `  print("Hello from a function")`

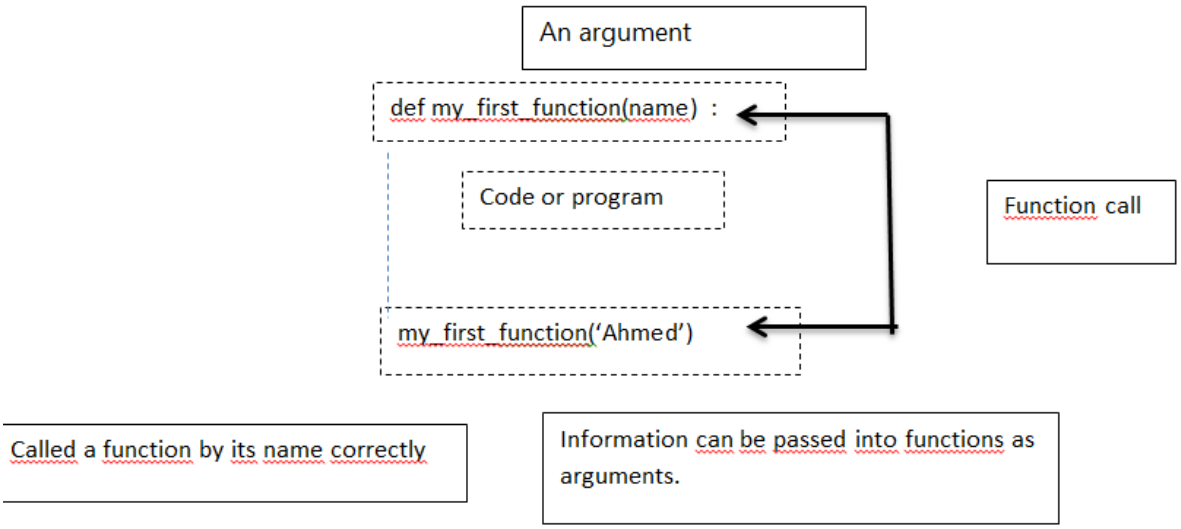
Calling a Function

To call a function, use the function name followed by parenthesis, the standard method involves using the function's identifier followed by parentheses.

General Syntax:

Code

```
functionName(parameter1, parameter2, ...)
```



Example

`def my_function():`
`  print("Hello from a function")`
 	
`my_function()`

`Hello from a function`

Arguments

Information can be passed into functions as arguments.

Arguments are specified after the function name, inside the parentheses. You can add as many arguments as you want, just separate them with a comma.

The following example has a function with one argument (fname). When the function is called, we pass along a first name, which is used inside the function to print the full name:

Example
def my_function(fname):   print(fname + " Refsnes") my_function("Emil") my_function("Tobias") my_function("Linus")
Emil Refsnes Tobias Refsnes Linus Refsnes

return statement

In Python, the **return** statement is used inside a function to send a value back to the caller and terminate the function's execution. It allows the result of a function's computation to be used in other parts of the program, promoting modular and reusable code.

examples of how to use the **return** statement:

```
# A function that returns a single value (an integer)
def add_numbers(a, b):
    result = a + b
    return result

# Store the returned value in a variable and use it
sum_value = add_numbers(5, 3)
print(f"The sum is: {sum_value}") # Output: The sum is: 8

# A function that returns a single value (an integer)
def add_numbers(a, b):
    result = a - b
    return result

# Store the returned value in a variable and use it
sub_value = add_numbers(5, 3)
print(f"The sub is: {sub_value}") # Output: The sub is: 2

# A function that returns a boolean value based on a condition
def is_even(num):
    return num % 2 == 0 # Returns True if the condition is met, False otherwise

print(f"Is 4 even? {is_even(4)}") # Output: Is 4 even? True
print(f"Is 5 even? {is_even(5)}") # Output: Is 5 even? False
```

```
The sum is: 8
The sub is: 2
Is 4 even? True
Is 5 even? False
```

Module

Consider a module to be the same as a code library.

A file containing a set of functions you want to include in your application.

using a function from a module, use the syntax: *module_name.function_name*.

Create a Module

To create a module just save the code in a file with the file extension .py:

Example

Save this code in a file named mymodule.py

```
def My_information(name):
  print("Hello, " + name)
```

Use a Module

Now we can use the module we just created, by using the import statement:

Example

Import the module named mymodule, and call the My_information function:

```
import mymodule

mymodule.My_information("Mohamed")
```

result

Hello Mohamed

Package

A **package** is a way of organizing related modules (single .py files) into a hierarchical directory structure. This structure, which uses "dotted module names" (e.g., package.submodule), helps manage large codebases, avoids naming conflicts, and promotes code reusability.

- A package is essentially a folder that contains an __init__.py file and one or more Python files (modules).
- Allows modules to be easily shared and distributed across different applications.

Create, import and, access packages

1-Create a Directory: Make a directory for your package. This will serve as the root folder.
2-Add Modules: Add Python files (modules) to the directory, each representing specific functionality.
3-Include __init__.py: Add an __init__.py file (can be empty) to the directory to mark it as a package.
4-Import Modules: Use dot notation to import, e.g., from mypackage.module1 import greet.

Exercices

Exercice 01

Write a function findHypot(a, b) that returns the length of the hypotenuse given the other two sides.

Exercice 02

Write a Python function to reverse a string.

Sample String: "1234abcd"
Expected Output: "dcba4321"

Exercice 03

Write a function is_even(n) that returns True if n is even, else False.

Write a function to check if a string is a palindrome (reads the same backward as forward).

Exercice 04

Write a function that returns:

- "fizz" if the number is divisible by 3,
- "buzz" if divisible by 5,
- "fizzbuzz" if divisible by both,
- otherwise an empty string.

Chapter 6: Listes, Tuples, and Sets

Before discussing listes,tuples, sets and dictionaries, we will introduce two important concepts: sequence types and mutability. A sequence type is a data type in Python that can store more than one value (or fewer than one, since a sequence can be empty). Mutuality is a property of all Python data that describes its ability to be freely modified during program execution.

The `for` loop is a tool specifically designed for iterating through sequences (dictionaries, tuples, and sets).

There are two types of Python data:

- Mutable: can be freely updated at any time
- Immutable: Immutable (or immutable) data cannot be modified.

Dictionaries are mutable, while tuples are immutable lists.

Lists (or list/arrays) in Python are variables that can hold multiple variables.

To create a list

```
liste = []
```

Add the values when creating the Python list:

```
liste = [1,2,3]

>>> liste

[1, 2, 3]
```

add after the list has been created using the append() method

```
>>> liste = []

>>> liste

[]

>>> liste.append(1)

>>> liste
```

```
[1]

>>> liste.append("ok")

>>> liste

[1, 'ok']
```

Accessing the values of a list

To access the values of a list, use square brackets [] to access an item by its index. The index must be an integer.

`liste = ["a","d","m"]`			
liste[0]	liste[2]	`tuple1` [0]	`tuple1` [0]
a	m		

The first item always starts with index 0. To read the first item, we use the value 0, the second, we use the value 1, and so on.

It is also possible to modify a value using its index.

`liste = ["a","d","m"]`			
liste[0]=”k”	liste[2]=”h”	`tuple1` [0]	`tuple1` [0]
`["k","d","m"]`	`["a","d","m"]`		

Deleting an entry with an index

To remove an entry from the list using the del function.

```
>>> liste = ["a", "b", "c"]

>>> del liste[1]

>>> liste

['a', 'c']
```

Removing

Removing an entry with its value

It is possible to remove an entry from a list along with its value using the remove () method.

```
>>> liste = ["a", "b", "c"]

>>> liste.remove("a")

>>> liste

['b', 'c']
```

List manipulation (methods)

Python includes methods for manipulating lists (reverse, len, count, index, in, etc.) see (table 1 and 2)

liste = [1, 2,3,1]		
Expression	Result	Description
len(liste)	4	Count the number of items in a list using the `len` function.
liste.reverse()	[1, 3, 2, 1]	Reverse the items in a list using the `reverse` method.
liste.count(1)	2	To find the number of occurrences of a value in a list, you can use the `count` method.
liste.index(2)	2	The `index` method allows to find the position of the item.
3 in liste	True	To check if an element is in a list, you can use the `in` keyword.
11 in liste	False	

```
liste = [1, 2,3,1]
print(len(liste))

liste.reverse()
print(liste)
print(liste.count(1))
print(liste.index(2))
print(3 in liste)
print(11 in liste)
```

```
4
[1, 3, 2, 1]
2
2
True
False
```

`liste = [1, 10, 100, 250, 500]`			
liste[-1]	liste[-3:]	liste[:]	liste[2:4]
500 Cherche la dernière occurence	[500, 250, 100, 10] Affiche les 3 dernières occurrences	[1, 10, 100, 250, 500] Affiche toutes les occurences	Tous les items compris enter l'index 2 et l'index 6

Looping through a list

To display the values of a list, you can use a loop (see chapter for loop):

```
>>> liste = ["a","d","m"]
>>> for lettre in liste:
...     print lettre
...
a
d
m
```

Tuples

Tuples in Python are ordered and immutable sequences, created by placing elements (of any type) separated by commas, usually between parentheses ().

Key features:

• Immutability: Once created, elements cannot be modified, added, or deleted.

• Order: Elements have a defined order that does not change.

• Mixed types: A single tuple can contain integers, strings, lists, etc.

Creation

Tuples are typically created with parentheses

```
# A simple tuple has nam Fruits
Fruits = ("apple", "banana", "cherry")
```

Accessing Elements

You access elements using square brackets [] and their index, or using negative indexing (where -1 refers to the last item).

```
fruits = ("apple", "banana", "cherry")
print(fruits[1])  # Output: 'banana'
print(fruits[-1]) # Output: 'cherry'
```

Allow Duplicates

Since tuples are indexed, they can have items with the same value:

```
thistuple = ("apple", "banana", "cherry", "apple", "cherry")
```

Operations

Despite being immutable, you can perform several operations:

- **Concatenation**: Combine two or more tuples using the + operator to create a new tuple.
- **Slicing**: Extract a subset of elements to form a new tuple using the slicing syntax [start:stop:step].

```
tuple1 = (1, 2)
tuple2 = (3, 4)
combined_tuple = tuple1 + tuple2 # (1, 2, 3, 4)
```

Tuple Methods

several built-in Python functions can be used with tuples for various operations:

count(): Returns the number of times a specified value occurs in a tuple.
Example (1, 2, 2, 3, 2).count(2) returns 3
index():Returns the index of the first occurrence of a specified value. An optional start and end index can be provided to limit the search within a specific range.
Example (10, 20, 30, 20).index(20) returns 1

- **len(tuple)**: Returns the total number of items in the tuple.
- **max(tuple)**: Returns the item from the tuple with the maximum value (elements must be of the same comparable type).
- **min(tuple)**: Returns the item from the tuple with the minimum value (elements must be of the same comparable type).
- **sum(tuple)**: Returns the sum of all numeric elements in the tuple.

Exercises

Exercise 1: Basic List Operations

1. Create a list of 5 fruits

2. Add 2 more fruits to the end

3. Remove the second fruit

4. Insert "banana" at position 1

5. Print the list and its length

Same operations with a list of products (your choice)

Exercise 2: List Manipulation

Given: numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

1. Create a new list with only even numbers

2. Create a new list with numbers squared

3. Find the sum, max, and min of the list

4. Reverse the list

Exercise 3: 2D Lists

Create a 3x3 matrix (list of lists) e.g a 2D array (nested list) called matrix:

```
matrix = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]
```

1. Access the element at row 2, column 3

2. Change the center element to 0

3. Print all elements using nested loops

4. Calculate the sum of all elements

5.Iterate through matrix and print each element row by row.

Tuples Exercises

Exercise 4: Basic Tuple Operations

1. Create a tuple of 5 colors

2. Try to modify the second element (observe error)

3. Convert tuple to list, modify, convert back to tuple

4. Create a tuple with single element

Exercise 5: Tuple Creation and Access

- Create a tuple called coordinates with values (10.5, 20.3).

- Print the entire coordinates tuple.

- Access and print the x-coordinate.

- Access and print the y-coordinate.

- Challenge: Try to change an element in coordinates and observe the error.

Exercise 6: Tuples in Loops

- Given a list of tuples students = [("Khaled", 15), ("Asma", 12), ("Fatima", 16)].

- Iterate through students and print each student's name and score in a formatted string (e.g., "Khaled scored 15").

Exercise 7

In this exercice we learn how manipulated the Complex Data Structures in Python for **Data Analysis project.**

Given data: temperatures = [("Jan", 5), ("Feb", 6), ("Mar", 10), ("Apr", 14), ("May", 18)]

1. Find the month with highest temperature

2. Calculate average temperature

3. Create a new list with temperatures converted to Fahrenheit

4. Find months with temperature above average

Exercise 8

Create a tuple of tuples: each containing month and its days

Example: (("January", 31), ("February", 28), ...)

1. Print all months with 31 days

2. Ask user for a month, show how many days it has

Chapter 7: Dictionaries

dictionary is a data structure that stores data in key-value pairs, where each key is unique and is used to retrieve its associated value. A dictionary is an unordered object containing multiple objects. Each object in a dictionary can be retrieved using its key. Dictionaries are mutable; that is, a dictionary can be updated by adding new values, modifying existing values, or deleting values.

A dictionary is a sequence of "key:value" pairs separated by commas and enclosed in curly braces.

Creating and Editing Dictionaries

Syntax for creating a dictionary following these four rules:

1- Each item describes a unique correspondence: unique key: value

2- A colon ":" must be used as the separator between the key and its corresponding value.

3- Items separated by commas.

4- The set of items enclosed between curly braces { }. { unique key: value, unique key: value, unique key: value, etc. }

Example

This example shows how a dictionary stores data of person using keys and values.

```
my_dict  = { "name": "Mohamed", "age": 22}
print(data)
```

Explanation:

- "name" and "age" are keys
- " Mohamed", 22, are their values

Add/Update dictionary

Use direct assignment with square brackets []. If the key exists, its value is updated; otherwise, a new key-value pair is added.

```python
my_dict  = { "name": "Mohamed", "age": 22}
print(my_dict)
```

```
{'name': 'Mohamed', 'age': 22}
```

```python

```

```python
my_dict = {"name": "Mohamed", "age": 22}
my_dict["name"] = "khaled"  # Update name
my_dict["age"] = 45         # Update age
print(my_dict)  # Output: {'name': 'khaled', 'age': 45}
```

```
{'name': 'khaled', 'age': 45}
```

Remove Item

Use the del keyword with the key name.

```
del my_dict["name"]
```

```python
my_dict  = { "name": "Mohamed", "age": 22}
print(my_dict)
```

```
{'name': 'Mohamed', 'age': 22}
```

```python
# Example 1: Delete 'name'
del my_dict["name"]
print( my_dict)  # {'age': 22}
```

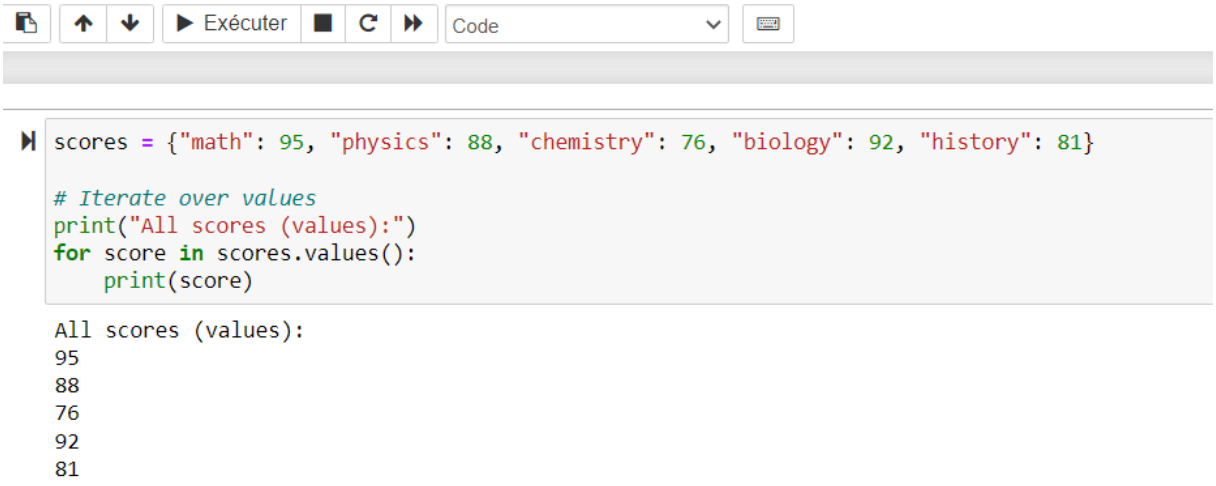
```
{'age': 22}
```

Dictionary Traversal Methods

The methods for traversing a Python dictionary involve using the methods like .items(), .keys(), and .values() within a for loop.

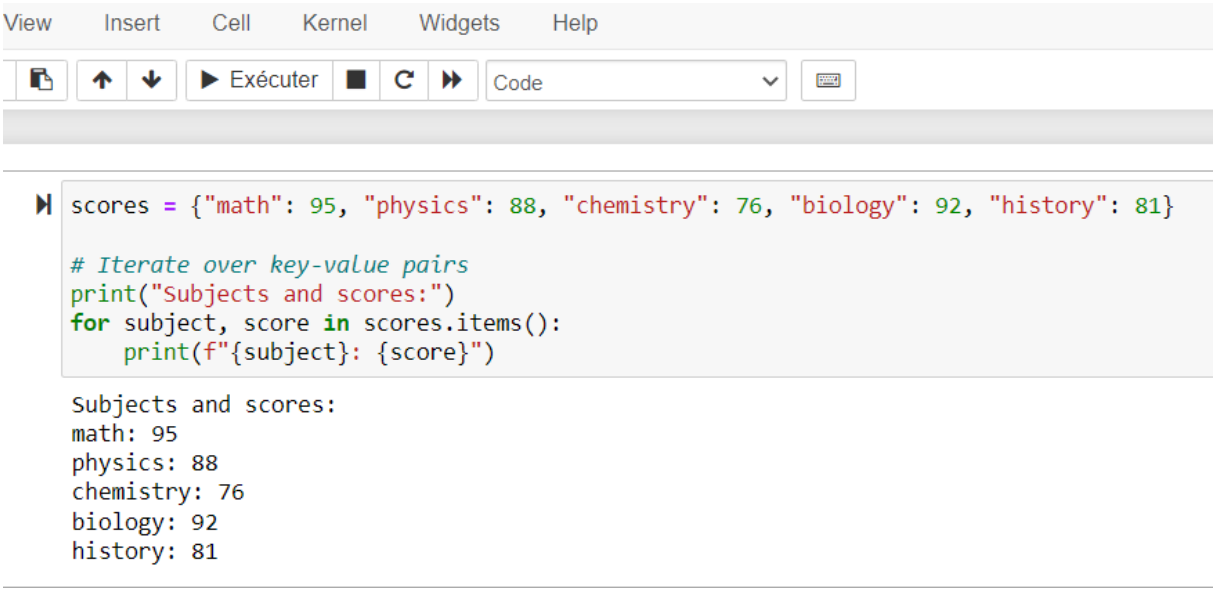
Iterate over values

Use the dict.values() method to iterate only over the values, the example Iterate over values and print each value.



Iterate over key-value pairs

1. Use the dict.items() method to iterate over both keys and values as tuples. This is the most common and efficient way. Examle Iterate over key-value pairs and print each key and value.



The .values() method gives you only the values, while .items() gives you both keys and values as tuples

```python
scores = {"math": 95, "physics": 88, "chemistry": 76, "biology": 92, "history": 81}

# 1. Filter keys during iteration
print("Subjects with scores > 85:")
for subject, score in scores.items():
    if score > 85:
        print(f"{subject}: {score}")

# 2. Iterate over specific keys only
print("\nSelected subjects:")
for subject in ["math", "physics", "biology"]:
    if subject in scores:
        print(f"{subject}: {scores[subject]}")

# 3. Skip certain keys
print("\nAll except history:")
for key, value in scores.items():
    if key != "history":
        print(f"{key}: {value}")
```

```
Subjects with scores > 85:
math: 95
physics: 88
biology: 92

Selected subjects:
math: 95
physics: 88
biology: 92

All except history:
math: 95
physics: 88
chemistry: 76
biology: 92
```

Dictionaries and Function Parameters

The dictionaries can be used with function parameters in Python.
There are two main concepts:

1. Using a dictionary as a parameter (i.e., passing a dictionary to a function)
2. Using **kwargs to accept arbitrary keyword arguments, which are then accessible as a dictionary.

Basic Dictionary as Parameter

The function receives the entire dictionary and accesses its elements using standard dictionary operations.

```python
def print_user_info(user_data):
    """Takes a dictionary as parameter"""
    print(f"Name: {user_data['name']}")
    print(f"Age: {user_data['age']}")
    print(f"Email: {user_data['email']}")

# Calling the function
user = {"name": "mohamed", "age": 25, "email": "mohamed@example.com"}
print_user_info(user)
```

```
Name: mohamed
Age: 25
Email: mohamed@example.com
```

Keyword Arguments (**kwargs)

uses ****kwargs** to accept any number of keyword arguments and prints them.

```python
def fun(**kwargs):
    for key, value in kwargs.items():
        print("key:", key, "Value:", value)

d1 = {'nom':"Mohamed", 'prenom': "benMohamed", 'age': 30}
fun(**d1)
```

```
key: nom Value: Mohamed
key: prenom Value: benMohamed
key: age Value: 30
```

Exercises

Exercise 1

Create an inventory dictionary that groups the products (the keys) and their quantities (the values).

```
inventaire = {'pommes': 100,'bananes': 80,'poires': 120}

inventaire.values()

inventaire.keys()
```

Execute the items() method which returns both the keys and the values

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

Exercise 2

Consider the following data, stored in the dictionary "information_telephone":

Name	telephone
Mohamed	0875421
Ahmed	0754218
Sirine	0784512
Sara	0784522

1. Create a "telephone_information" dictionary for this data.

2. How do I know if "Mohamed" is registered?

3. Change Ahmed's number; it ends in a 9, not an 8.

4. Add "Khaled," whose number is "0721453."

5. Remove "Sara" from the directory.

Exercise 3

Write a function "fusion_dict(dict1, dict2)" that merges two dictionaries. If a key exists in both, keep the value from the second dictionary.

Example :
>>> fusion_dict({"a": 1, "b": 2}, {"b": 3, "c": 4})
{"a": 1, "b": 3, "c": 4}

Chapiter 8 : Object-Oriented Programming (OOP)

Object-oriented programming is a paradigm, that is, a way of writing clear and simple programs. The principle is to model the elements of program (such as arrays and lists) as objects characterized by attributes and capable of performing actions. These objects are built from classes that contain their blueprint.

In Python, almost everything is designed to be an object: lists, dictionaries, NumPy arrays, and so on. For example, when you write `list.append()`, you are actually using the `append()` method on a list object. Therefore, the documentation for Python, NumPy, Pandas, Matplotlib, and Sklearn is largely composed of classes that are important to understand in order to learn new things independently using the documentation.

Object-oriented design involves thinking in terms of objects and their interactions.

- A system is broken down into a set of objects with clearly defined roles.
- Objects collaborate and interact to achieve the overall functionality of the system.
- Principles such as encapsulation, inheritance, and polymorphism are used to capture similarities and relationships between real-world entities in software design.

Object-Oriented Programming (OOP) Development Process

1. Analysis Phase

- **Identify real-world entities** (objects) in the problem domain
- **Define responsibilities** of each object
- **Identify relationships** between objects (association, aggregation, composition)
- **Gather requirements** and use cases
- Create **conceptual models**

2. Design Phase

- **Class Design**:
  - Identify classes and their attributes (data)
  - Define methods (behavior)

3.Relationships Design:

  - Inheritance hierarchies
  - Interface implementations
  - Association relationships
  - Dependency injections
- **Design Patterns** (when applicable):
  - Creational (Factory, Singleton, Builder)

- Structural (Adapter, Decorator, Facade)
- Behavioral (Observer, Strategy, Command)

Basic Concepts – Classes and objects

♦ A class provides an abstraction by defining a generic model for objects. It allows developers to focus on relevant aspects without concerning themselves with complex details.

♦ **A class** is an entity that groups data (attributes) and operations/functions (methods) associated with a set or group of objects sharing the same structure. Classes provide a means of bundling data and functionality together.

- Attributes, also called properties, represent the characteristics or data that each object of the class will possess. These attributes describe the state of the object.
- Methods are functions associated with the class that specify the behavior of its objects. They define the actions that objects can perform

"Class" is the structure e.g a class is a structured type that goes beyond records, lists, dictionnary, etc. because group together in the same structure (***data (attributes) and operations/functions)***

Object : is an instance of the class (a variable obtained after instantiation). When a class is defined, no memory is allocated until an object of that class is created

```
#definition

class Name_of_class:

"""name class"""

# constructor

def __init__(self):

# list the fields

self.firstfields = …

self.scondfeilds = …

self.thirdfields = …

#fin constructeur
```

`class` is a keyword used to define the structure.

• `Name_of_Class` is the name of the class here.

• `"" Name_of_Class """` is used to document the class e.g comments.

• Note the role of colons (`:`) and indentation.

• `__init__` is a standard method called a constructor, automatically called during instantiation.

• `self` represents the instance itself; it must appear first in the definition of all methods, but it will not be necessary to specify it when calling them.

Create a Person class with:

• A constructor that takes name and age as parameters

• A method `se_presenter()` that displays "My name is [name] and I am [age] years old"

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def display(self):
        print(f"my name is {self.name} and I am {self.age} ans")
```

```
p1 = Person("Mohamed", 25) # creat object p1
p1.display()  # Should display: My name is Mohamed and I am 25 years old
```

```
my name is Mohamed and I am 25 ans
```

```
p2 = Person("khaled", 33) # creat object p2
p2.display()  # Should display: My name is khaled and I am 33 years old
```

```
my name is khaled and I am 33 ans
```

An example of a Python class representing a book, complete with attributes and methods.

```
class Book:
    """
    Represents a book with a title, author, and publication year.
    """
    def __init__(self, title, author, year):
        """
        Initializes the Book object with the provided title, author, and year.
        The __init__ method is automatically called when a new instance is created.
        """
        self.title = title
        self.author = author
        self.year = year

    def get_age(self, current_year):
        """
        Calculates and returns the age of the book in years.
        """
        return current_year - self.year

    def display_info(self):
        """
        Prints information about the book.
        """
        print(f"Title: {self.title}")
        print(f"Author: {self.author}")
        print(f"Publication Year: {self.year}")

    def __str__(self):
        """
        Special method called when the print function or str() is used on the object.
        Returns a string representation for easy reading.
        """
        return f"'{self.title}' by {self.author}"
```

```
# --- Example Usage ---

# 1. Create an instance (object) of the Book class
book1 = Book("The Hitchhiker's Guide to the Galaxy", "Douglas Adams", 1979)

# 2. Access attributes
print(f"The author of book1 is: {book1.author}")

# 3. Call methods
book1.display_info()

# 4. Use the __str__ method via print()
print(f"String representation: {book1}")

# 5. Calculate the book's age
current_year = 2025 # Using the current year for calculation
age = book1.get_age(current_year)
print(f"The book is {age} years old.")
```

Chapiter 9 : File Handling In Python

In programming, a file refers to a unit of storage on secondary devices such as a disk or solid-state drive, where information is preserved in a non-volatile manner, ensuring that data remains intact even when the power is off.

File handling refers to the process of performing operations on a file, such as creating, opening, reading, writing and closing it through a programming interface.

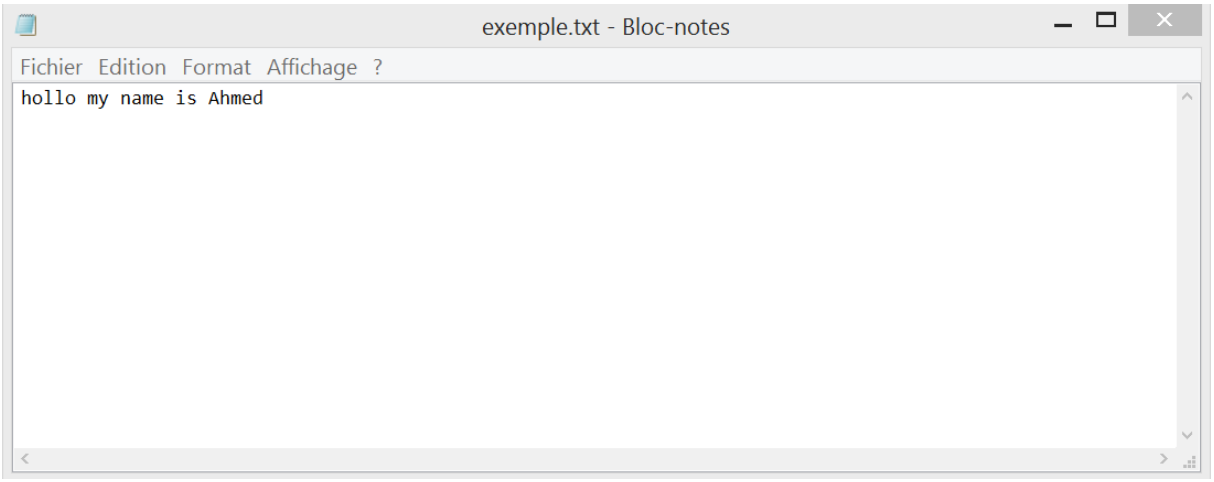
Advantages of using files

Using files in Python provides essential capabilities that make programs more useful, efficient, and professional. Here's a breakdown of the key advantages, from core functionality to practical benefits.

Advantage	Description & Key Benefit	Python Feature / Example
Data Persistence	Saves program data permanently (on disk) so it exists after the program closes. Enables apps that remember information.	Writing to .txt, .csv, .json files.
Handling Large Datasets	Processes data too large for RAM by reading/writing in chunks. Enables work with massive logs, datasets, etc.	Reading files line-by-line (for line in file:).
Interoperability & Data Exchange	Reads input from and writes output to standard formats for exchange with other systems (web APIs, databases, other programs).	csv module for spreadsheets, json module for web data.
Configuration & Initialization	Stores settings (API keys, user preferences) outside the code. Changes settings without modifying the program.	Reading from a config.ini or settings.json file.
Logging & Auditing	Creates permanent records of program events, errors, and user actions for debugging, monitoring, and audit trails.	Writing timestamps and messages to a .log file.

File Formats (TXT, CSV, JSON,XLSX, XML)

Several commonly used file formats in data science and artificial intelligence: **TXT, CSV, XLSX, JSON, and XML.**



CSV FILE

It is a structured text format used for storing tabular data, where each line corresponds to a row and each value is separated by a comma.

EXCEL FILE

XLSX is the standard format for Microsoft Excel files, used to store complex spreadsheet data including formulas, formatting, and multiple sheets. Unlike the plain-text formats above, XLSX is a binary format that requires libraries like openpyxl or pandas in Python for access.

JSON FILE

JSON is a standard text-based format for representing structured data based on JavaScript object syntax. It supports structured data and is both human and machine-readable.

XML FILE

XML (eXtensible Markup Language) is a markup language used to encode documents in a structured and hierarchical format. It is often used in data exchange protocols and web services

File Creation and Management

File creation and management in Python involves the entire lifecycle—from creating and writing to organizing and deleting files. Here's a practical guide covering the core operations, best practices, and useful modules.

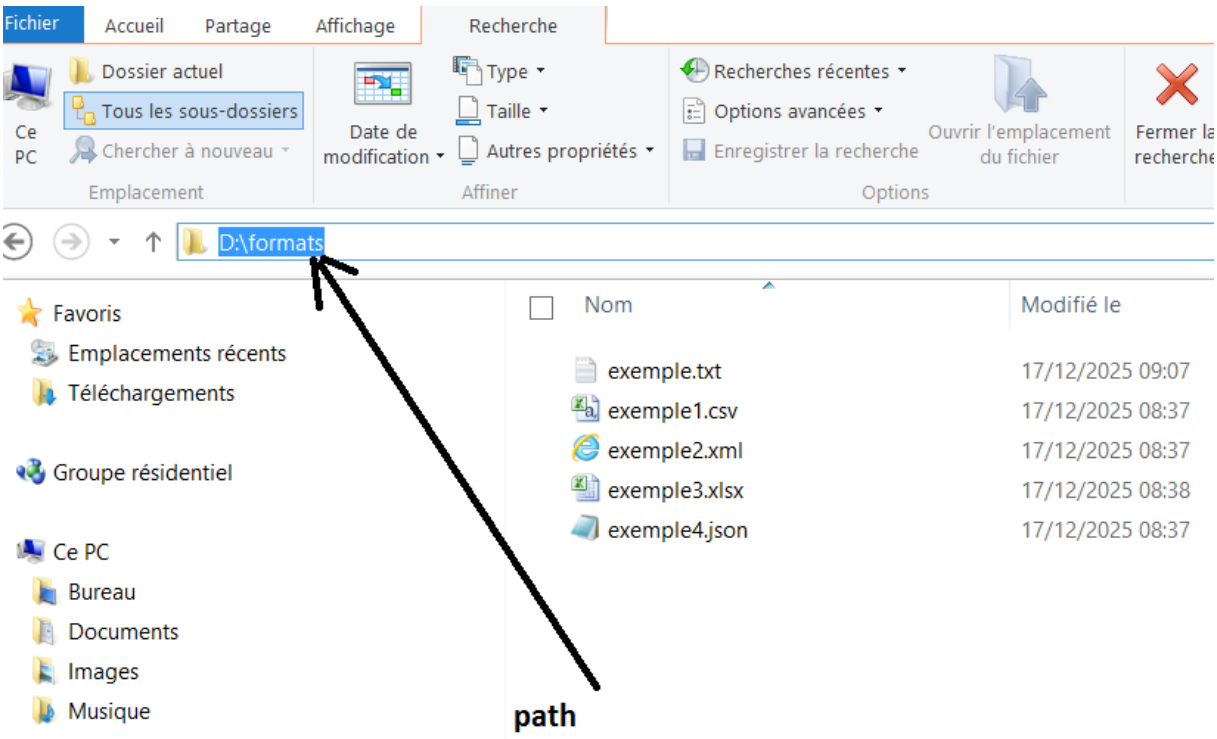
Open a file in Python

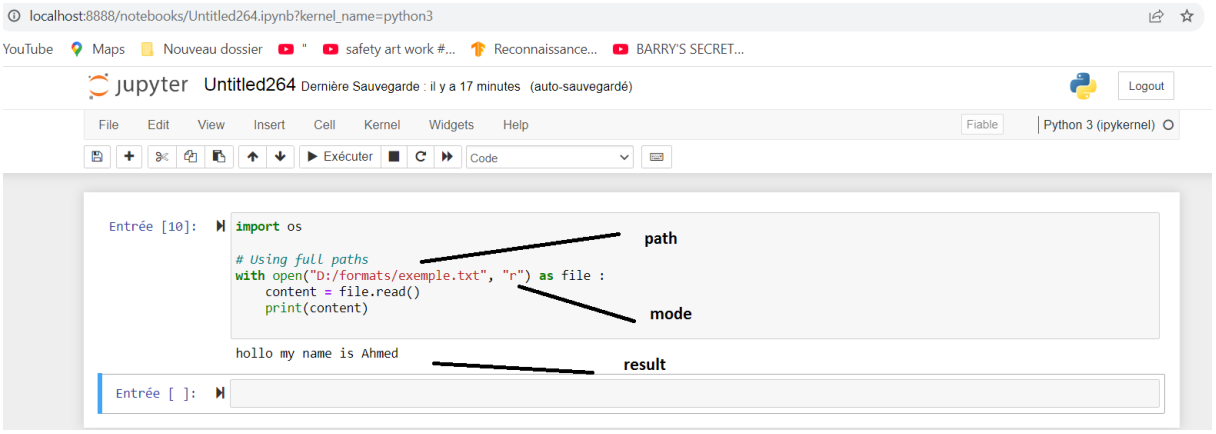
The open() function requires two main arguments: the file path and the mode. A file path consists of two main parts: the directory path and the file name. The mode defines the operation to be performed on the file, such as 'r' for reading, 'w' for writing, 'a' for appending, or 'r+' for reading and writing.

file = open('filename.txt', 'mode')

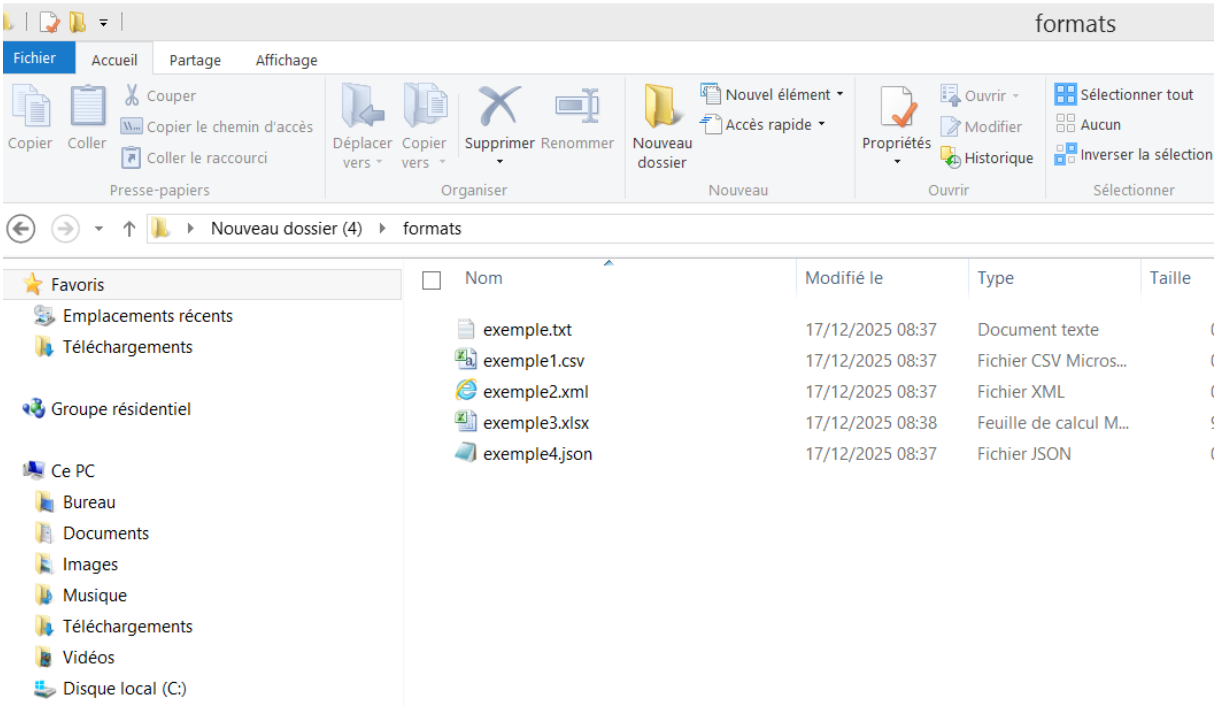
filename.txt: name (or path) of the file to be opened.

mode: mode in which you want to open the file (read, write, append, etc.).





These operations use Python's built-in open() function and various modes.



TEXT FILE

The TXT format represents the most basic form of a text file, typically used for unstructured or lightly structured human-readable content. For example, a .txt file may simply contain a list of names or paragraphs of text.

Opening Modes & Core Methods

Mode	Description	Use Case
'r'	Read (default). Opens for reading.	Reading existing files. Error if file doesn't exist.
'w'	Write. Creates new or overwrites existing.	Creating new files or replacing content.
'a'	Append. Creates new or adds to end of existing.	Logging, adding data without changing old content.

as arguments. The syntax is file_object = open(filename, mode).

with statement: The recommended best practice for handling files. It automatically ensures the file is closed.

1. Writing to a file

Use the write() or writelines() methods to add content to a file.

```
# Open a file in write mode ('w') and write a single line
with open("example.txt", "w") as file:
    file.write("Hello, World!\n")
    file.write("This is a second line.")
```

2. Reading from a file

Use methods like read(), readline(), or readlines() to get data from a file.

```
# Open a file in read mode ('r') and read the entire content
with open("example.txt", "r") as file:
    content = file.read()
    print(content)
# Looping through a file line by line
with open("example.txt", "r") as file:
    for line in file:
        print(line, end='') # 'end="" prevents extra newlines
```

Useful Operations

For managing files and directories (renaming, deleting, checking existence), you can use the built-in os module.

```
import os

# Check if a file exists

if os.path.exists("example.txt"):

    # Delete a file

    os.remove("example.txt")

else:

    print("File does not exist.")

# Rename a file

os.rename("old_name.txt", "new_name.txt")
```

CSV, JSON, or Excel files

For handling structured data like CSV, JSON, or Excel files, Python provides specific modules such as csv, json, and external libraries like openpyxl or pandas. Then use the same operations and techniques as with text files.

Exercices

Exercise 1

Write a Python program that :

- Creates a new text file named hello.txt and writes the line "Hello, World!" into it.
- Opens the file hello.txt and prints its contents to the console.
- Adds the line "This is a new line." to the end of the file. Then read and display the updated file.
- counts the total number of words in it

Exercise 2

Write a Python script that performs the following steps:

1. Create a new Excel file named students.xlsx.

2. Add a worksheet containing student names and their scores in two subjects (e.g., Math and Science).

3. Write at least 5 rows of data.

4. Re-open the same file and:

• Read all rows.

• Calculate the average score for each student.

• Print the name and average score

Exercise 3

Given the table bollow containing basic information about three Students, including their first Name and Last Name, write a Python script that represents this data with suitable data structures (lists and dictionaries) and saves it to two separate files: one in csv format (Student.csv) and another in excel format (students.xlsx).

Id_Student	Firset name	Laste name
1	mohamed	benmohamed
2	ali	benali
3	jamel	benjamel

Exercise Solutions

Exercise Solutions in chapter 2

Exercise 1

Insert the parentheses and give the result

1-2*5**3

In Python, the expression `1 - 2 * 5 ** 3` is evaluated according to operator precedence: exponentiation (`**`) has the highest precedence, followed by **multiplication** (`*`), and then subtraction (`-`). So the implicit grouping is:

1 - (2 * (5 ** 3))=- **249**

If you insert parentheses differently, the result changes. For example:

- (1 - 2) * 5 ** 3 ⟶ (-1) * 125 = **-125**
- 1 - (2 * 5) ** 3 ⟶ 1 - 10 ** 3 = 1 - 1000 = **-999**

But without any explicit parentheses, Python uses the default precedence, giving **-249**.

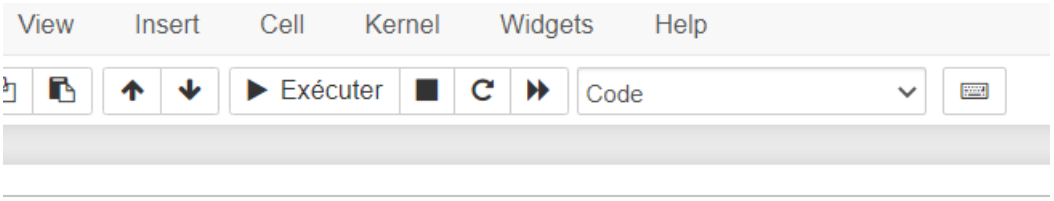
What placeholders are used when dealing with f-strings?

() ;{} , ||, ‘ ‘ or []

F-strings in Python use curly braces {} as placeholders to embed variables or expressions directly into string literals. For **purpose** to directly insert variables, expressions, or function calls into the string,

Example: `name = "Mohamed"; print(f"Hello, {name}!")`

Exercise 4 (Logic)



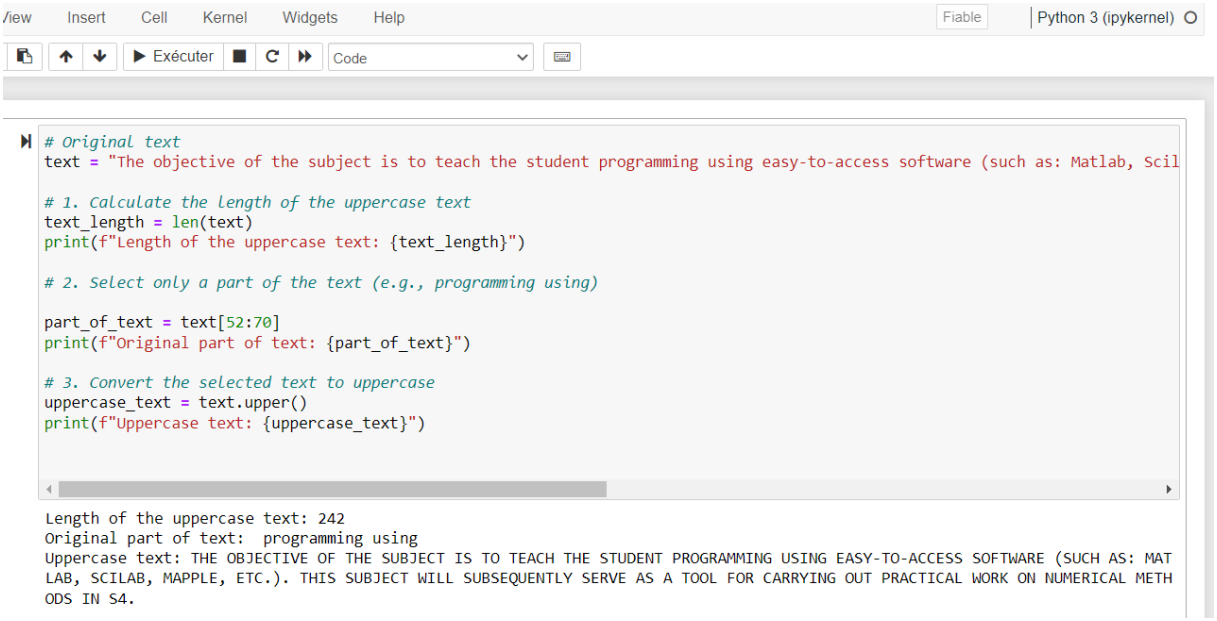

```
a=0
b=1
c=a==1
print(c)
d=not(b) # ~b negation of b
print(d)
print(a<b)
print(a!=b)
e=not(c or d)  #e=~(c|d)
print("e is : ",e)
e1=(c and d) #e1=(c & d)
print("e1 is :",e1)
```

```
False
False
True
True
e is :  True
e1 is : False
```

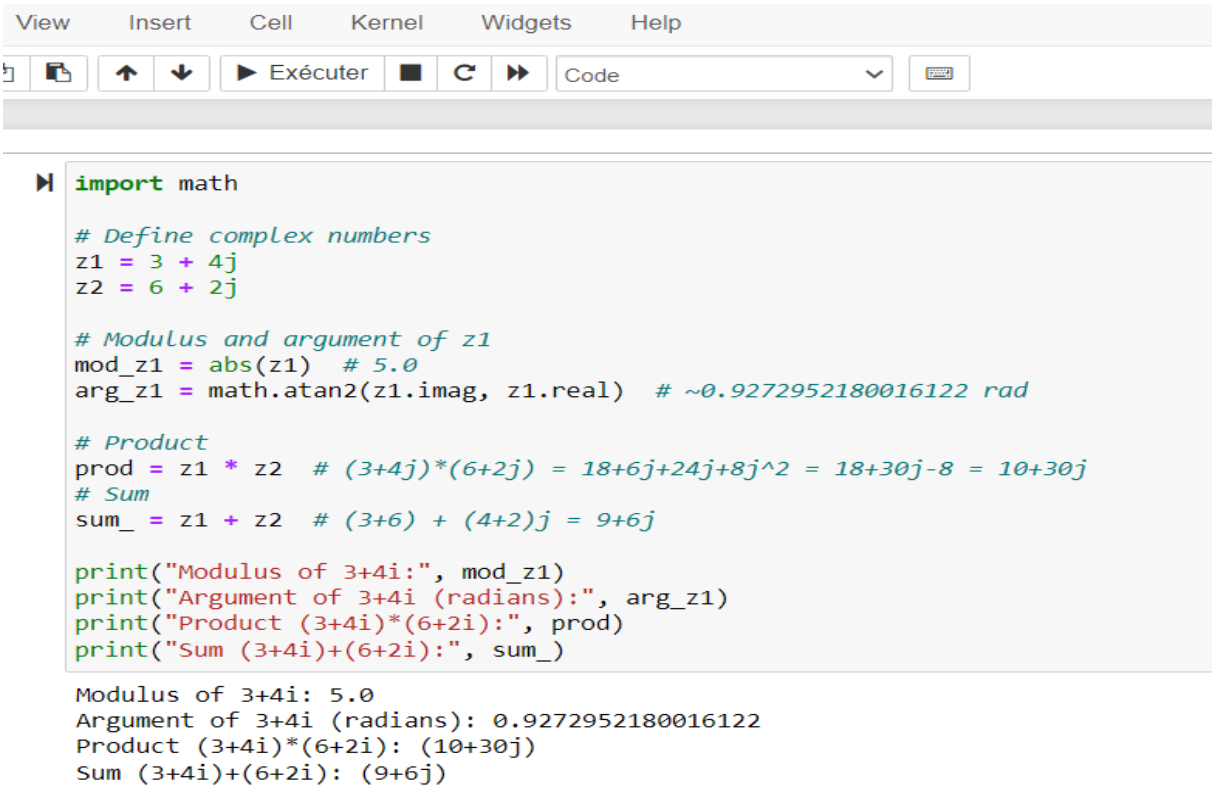
Exercise 5 (Character String)

```
# Original text
text = "The objective of the subject is to teach the student programming
using easy-to-access software (such as: Matlab, Scilab, Mapple, etc.). This
subject will subsequently serve as a tool for carrying out practical work
on numerical methods in S4."


# 1. Calculate the length of the uppercase text
text_length = len(text)
print(f"Length of the uppercase text: {text_length}")


# 2. Select only a part of the text (e.g., programming using)

part_of_text = original_text[52:70]
print(f"Original part of text: {part_of_text}")


# 3. Convert the selected text to uppercase
```

```
uppercase_text = text.upper()
print(f"Uppercase text: {uppercase_text}")
```



Exercise 6 (complex number)



Exercise Solutions in chapter 3

Exercise 1

```
num = float(input("Enter a number: "))

    # Check the number
    if num > 0:
        print(f"{num} is positive.")
    elif num < 0:
        print(f"{num} is negative.")
    else:
        print("The number is zero.")
```

Exercise 2

```
num = int(input("Enter a number: "))

    if num == 7:
        print("AA")
    elif num == 80:
        print("BB")
    elif num == 100:
        print("CC")
    elif num == 250:
        print("DD")
    else:
        print("ZZAA")
```

Exercise 3

```
age = int(input("Enter the child's age: "))

    if age < 0:
        print("Age cannot be negative.")
    elif age < 6:
        print("Too young for these categories.")
    elif age <= 7:
        print("Poussin")
    elif age <= 9:
        print("Pupille")
    elif age <= 11:
        print("Minime")
    else:
        print("Cadet")
```

Exercise 4

```
import math
print("Quadratic equation solver: ax^2 + bx + c = 0")
a = float(input("Enter coefficient a: "))
b = float(input("Enter coefficient b: "))
c = float(input("Enter coefficient c: "))
if a == 0:
    print("Coefficient a is zero. Solving linear equation:")
    if b == 0:
        if c == 0:
            print("Infinite solutions (0 = 0).")
        else:
            print("No solution (contradiction).")
    else:
        x = -c / b
        print(f"One real root: x = {x:.4f}")

# Quadratic case
discriminant = b**2 - 4*a*c
if discriminant > 0:
    sqrt_d = math.sqrt(discriminant)
    x1 = (-b - sqrt_d) / (2*a)
    x2 = (-b + sqrt_d) / (2*a)
    print(f"Two distinct real roots: x1 = {x1:.4f}, x2 = {x2:.4f}")
elif discriminant == 0:
    x = -b / (2*a)
    print(f"One double real root: x = {x:.4f}")
else:
    real_part = -b / (2*a)
    imag_part = math.sqrt(-discriminant) / (2*a)
    print(f"Two complex conjugate roots:")
    print(f"x1 = {real_part:.4f} + {imag_part:.4f}i")
    print(f"x2 = {real_part:.4f} - {imag_part:.4f}i")
```


```python
import math
print("Quadratic equation solver: ax^2 + bx + c = 0")
a = float(input("Enter coefficient a: "))
b = float(input("Enter coefficient b: "))
c = float(input("Enter coefficient c: "))
if a == 0:
    print("Coefficient a is zero. Solving linear equation:")
    if b == 0:
        if c == 0:
            print("Infinite solutions (0 = 0).")
        else:
            print("No solution (contradiction).")
    else:
        x = -c / b
        print(f"One real root: x = {x:.4f}")

# Quadratic case
discriminant = b**2 - 4*a*c
if discriminant > 0:
    sqrt_d = math.sqrt(discriminant)
    x1 = (-b - sqrt_d) / (2*a)
    x2 = (-b + sqrt_d) / (2*a)
    print(f"Two distinct real roots: x1 = {x1:.4f}, x2 = {x2:.4f}")
elif discriminant == 0:
    x = -b / (2*a)
    print(f"One double real root: x = {x:.4f}")
else:
    real_part = -b / (2*a)
    imag_part = math.sqrt(-discriminant) / (2*a)
    print(f"Two complex conjugate roots:")
    print(f"x1 = {real_part:.4f} + {imag_part:.4f}i")
    print(f"x2 = {real_part:.4f} - {imag_part:.4f}i")
```

```
Quadratic equation solver: ax^2 + bx + c = 0
Enter coefficient a: 1
Enter coefficient b: 20
Enter coefficient c: 3
Two distinct real roots: x1 = -19.8489, x2 = -0.1511
```

Exercise 5

```
year = int(input("Enter a year: "))

    # Leap year conditions
    if (year % 4 == 0 and year % 100 != 0) or (year % 400 == 0):
        print(f"{year} is a leap year.")
    else:
        print(f"{year} is not a leap year.")
```

Exercise Solutions in chapter 4

Exercise1 (For loop)

```
sum_numbers = 0  # Initialize sum to zero

# Loop from 1 to 100 (inclusive)
for i in range(1, 101):
    sum_numbers += i

# Display the result
print("The sum of all numbers from 1 to 100 is:", sum_numbers)
```

Exercise 2 :Loop Control Statements (break and continue)

```
# Find the first prime number greater than 100 using break
num = 101
while True:
    # Check if num is prime
    is_prime = True
    for i in range(2, int(num**0.5) + 1):
        if num % i == 0:
            is_prime = False
            break
    if is_prime:
        print(f"First prime number greater than 100: {num}")
        break
    num += 1

# Print numbers from 1 to 20 except multiples of 3 using continue
print("\nNumbers from 1 to 20 excluding multiples of 3:")
for i in range(1, 21):
    if i % 3 == 0:
        continue
    print(i, end=" ")
print()  # new line
```

Exercise 3 :While loop/do-While loop

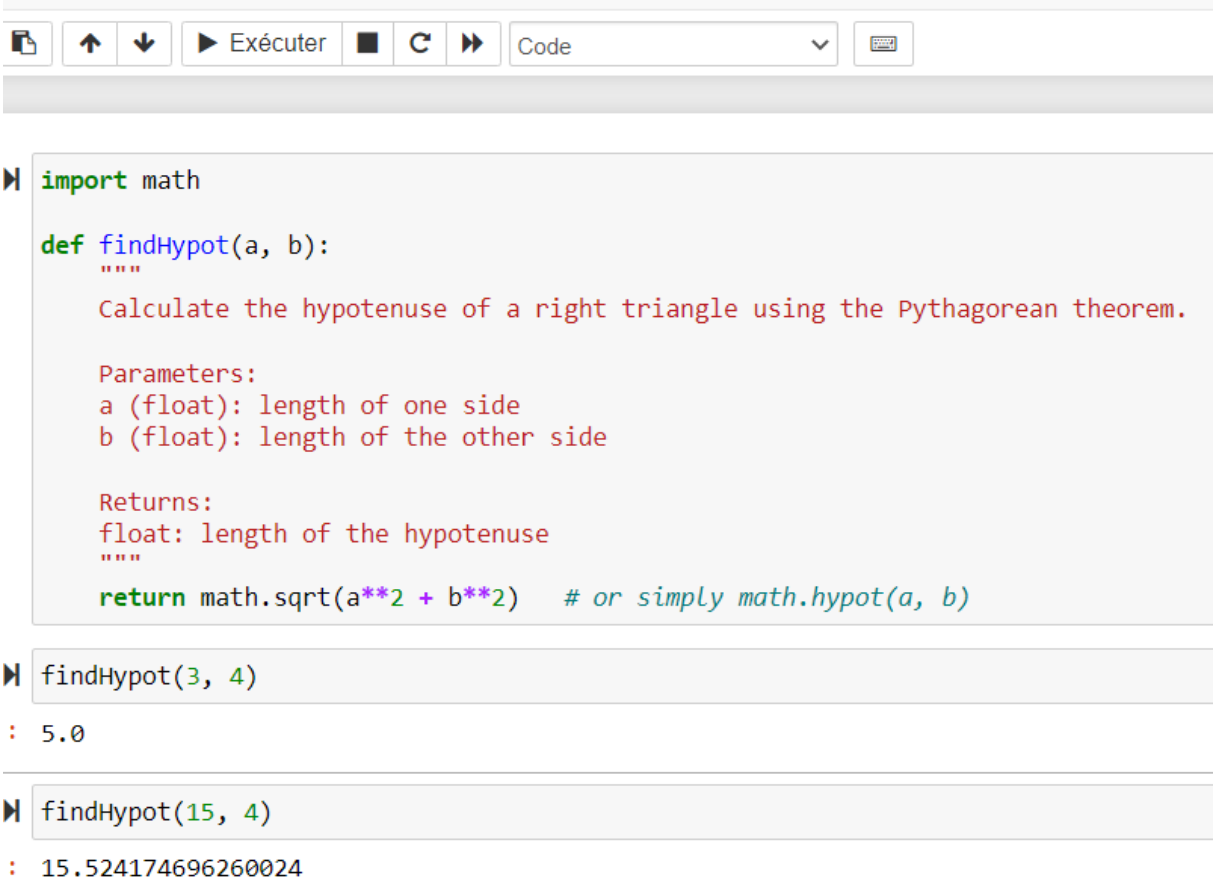
```
total = 0
count = 0

while True:

        num = int(input("Enter an integer (0 to stop): "))
        continue

    if num == 0:
        break

    total += num
    count += 1

if count == 0:
    print("No numbers were entered (excluding zero).")
else:
    average = total / count
    print(f"Average of the entered numbers: {average:.2f}")
```

Exercise 4

```
# Program to remove all letters 'a' and 's' from user input

text = input("Enter text: ")

result = ""
for char in text:
    if char != 'a' and char != 's':
        result += char

print(result)
```

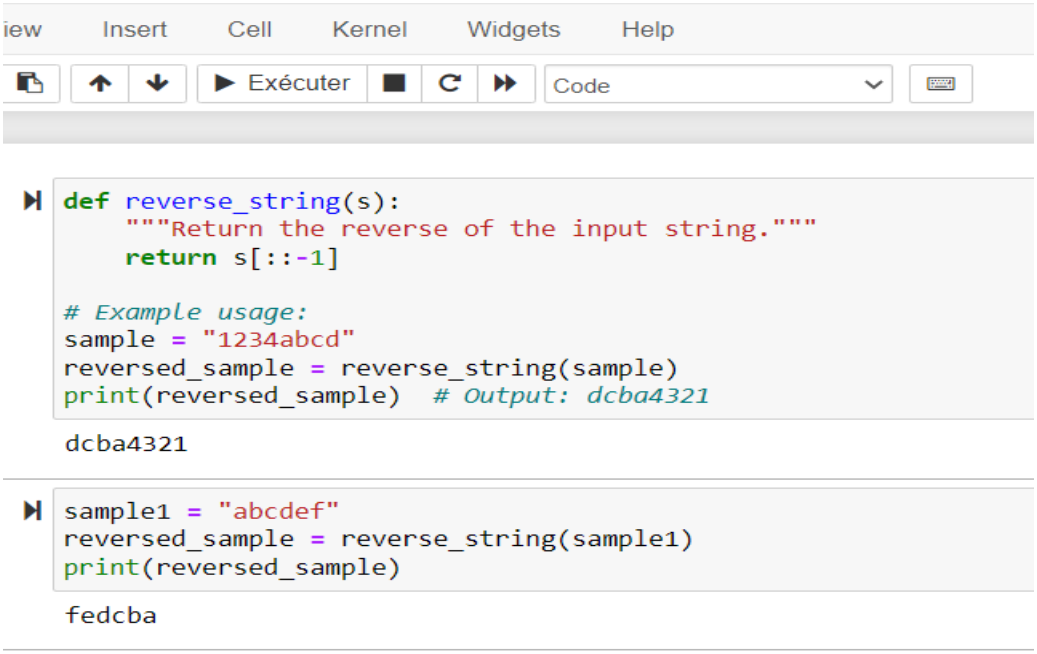
Exercise Solutions in chapter 5

Exercice 01

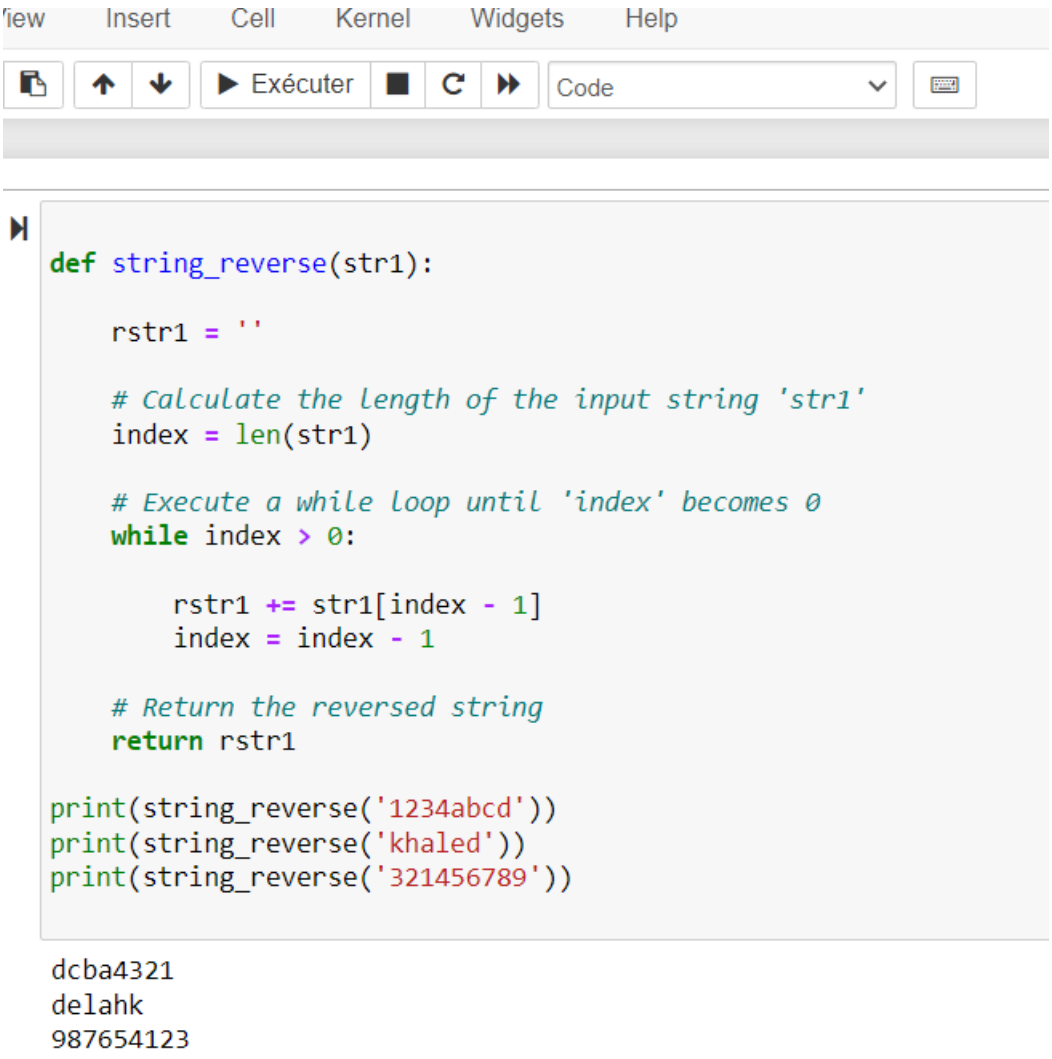


Exercice 02

Python function that reverses a string using slicing:



Python function that reverses a string using while loop



Exercice 03

```
def is_even(n):
    """Return True if n is even, False otherwise."""
    return n % 2 == 0

def is_palindrome(s):
    """Return True if string s is a palindrome, False otherwise."""
    return s == s[::-1]

# Example usage:
print(is_even(4))    # True
print(is_even(7))    # False

print(is_palindrome("racecar"))  # True
print(is_palindrome("hello"))    # False
```

Exercice 04

This function checks divisibility in the correct order, giving priority to the combined

case first.

```
def fizzbuzz(n):
    """Return 'fizz' if divisible by 3, 'buzz' if divisible by 5,
    'fizzbuzz' if divisible by both, otherwise empty string."""
    if n % 3 == 0 and n % 5 == 0:
        return "fizzbuzz"
    if n % 3 == 0:
        return "fizz"
    if n % 5 == 0:
        return "buzz"
    return ""

# Example usage:
print(fizzbuzz(3))   # fizz
print(fizzbuzz(5))   # buzz
print(fizzbuzz(15))  # fizzbuzz
print(fizzbuzz(7))   # (empty string)
```

Exercise Solutions in chapter 6

Exercise 1: Basic List Operations

```
  # 1. Create a list of 5 fruits
fruits = ['apple', 'orange', 'grape', 'mango', 'pear']
print("Initial fruit list:", fruits)

# 2. Add 2 more fruits to the end
fruits.append('kiwi')
fruits.append('pineapple')
print("After adding two fruits:", fruits)

# 3. Remove the second fruit (index 1)
#    (the list is now: apple, orange, grape, mango, pear, kiwi, pineapple)
removed_fruit = fruits.pop(1)   # removes 'orange'
print(f"Removed the second fruit: '{removed_fruit}'")
print("After removal:", fruits)

# 4. Insert "banana" at position 1 (which becomes the new second element)
fruits.insert(1, 'banana')
print("After inserting 'banana' at position 1:", fruits)

# 5. Print the list and its length
print("Final fruit list:", fruits)
print("Length of fruit list:", len(fruits))
print()

# --------------------------
# Part 2: Same operations with a list of products (your choice)
# --------------------------

# 1. Create a list of 5 products
products = ['laptop', 'mouse', 'keyboard', 'monitor', 'headphones']
print("Initial product list:", products)

# 2. Add 2 more products to the end
products.append('webcam')
products.append('USB cable')
print("After adding two products:", products)

# 3. Remove the second product (index 1)
removed_product = products.pop(1)   # removes 'mouse'
print(f"Removed the second product: '{removed_product}'")
print("After removal:", products)

# 4. Insert "tablet" at position 1
products.insert(1, 'tablet')
print("After inserting 'tablet' at position 1:", products)

# 5. Print the list and its length
print("Final product list:", products)
print("Length of product list:", len(products))
```

Exercise 2: List Manipulation

```
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

# 1. Even numbers
evens = [num for num in numbers if num % 2 == 0]
print("Even numbers:", evens)

# 2. Squares
squares = [num ** 2 for num in numbers]
print("Squares:", squares)

# 3. Sum, max, min
total = sum(numbers)
maximum = max(numbers)
minimum = min(numbers)
print("Sum:", total, "Max:", maximum, "Min:", minimum)

# 4. Reverse the list
reversed_list = numbers[::-1]  # or list(reversed(numbers))
print("Reversed:", reversed_list)
```

Exercise 3: 2D Lists

```
# Create a 3x3 matrix (list of lists)
matrix = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]

element = matrix[1][2]
print("Element at row 2, column 3:", element)  # Output: 6

# 2. Change the center element to 0
#    Center is row 2, column 2 (index [1][1])
matrix[1][1] = 0
print("Matrix after changing center to 0:")
for row in matrix:
    print(row)

# 3. Print all elements using nested loops
print("\nAll elements printed using nested loops:")
for i in range(len(matrix)):          # iterate over rows
    for j in range(len(matrix[i])):   # iterate over columns in current row
        print(f"matrix[{i}][{j}] = {matrix[i][j]}")

# 4. Calculate the sum of all elements
total = 0
```

```
for row in matrix:
    total += sum(row)   # sum each row and add to total
print("\nSum of all elements:", total)

# 5. Iterate through matrix and print each element row by row
print("\nPrinting each element row by row:")
for row in matrix:
    for element in row:
        print(element, end=" ")   # print elements in the same row with space
    print()                       # new line after each row
```

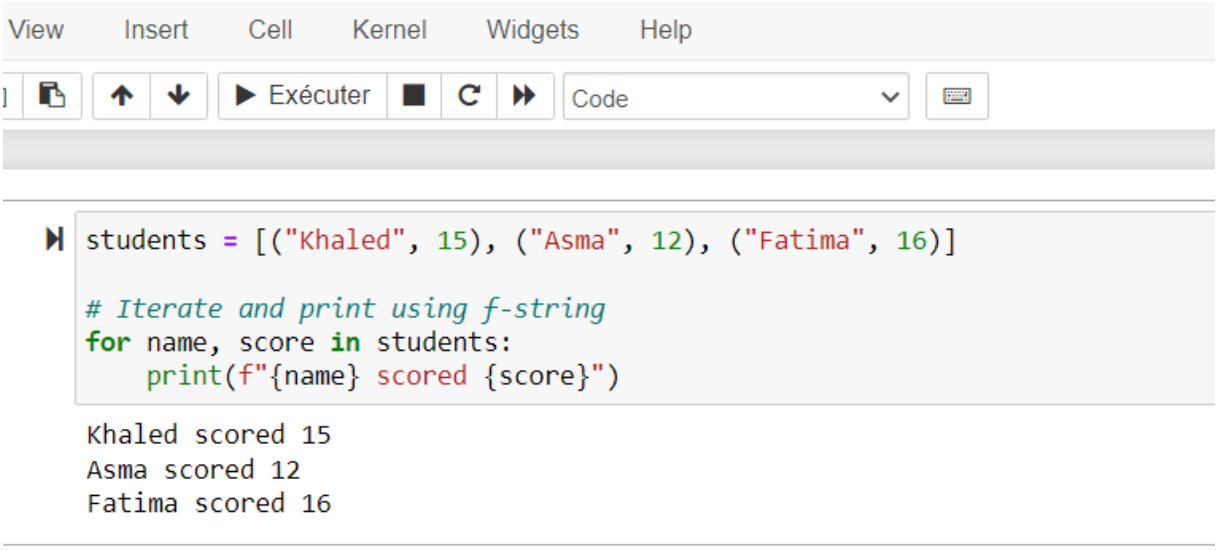
Tuples Exercises

Exercise 4: Basic Tuple Operations

```
  # 1. Create a tuple of 5 colors
colors = ('red', 'green', 'blue', 'yellow', 'purple')
print("Original tuple:", colors)

colors[1] = 'orange'   # Tuples are immutable

# 3. Convert tuple to list, modify, convert back to tuple
temp_list = list(colors)   # convert to list
temp_list[1] = 'orange'    # modify the second element
modified_colors = tuple(temp_list)  # convert back to tuple
print("\nModified tuple (after conversion):", modified_colors)
```

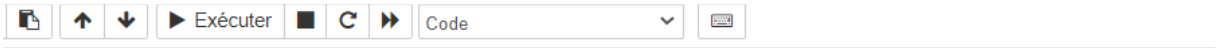
Exercise 6: Tuples in Loops

Objective is iterate through the list of tuples and print formatted strings:



Exercise 7

In this exercice we learn how manipulated the Complex Data Structures in Python for Data Analysis project.



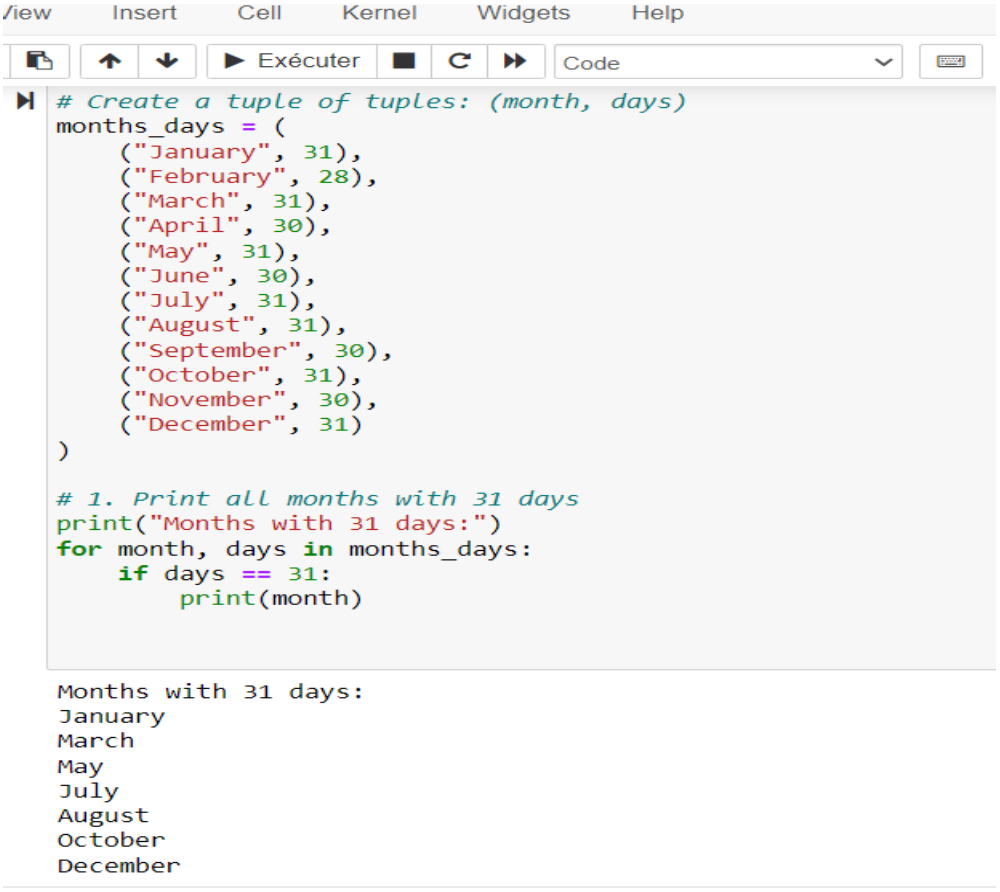

```python
# Given data
temperatures = [("Jan", 5), ("Feb", 6), ("Mar", 10), ("Apr", 14), ("May", 18)]

# 1. Find the month with highest temperature
# Use max() with a key that looks at the temperature (second element)
highest_month, highest_temp = max(temperatures, key=lambda item: item[1])
print(f"1. Month with highest temperature: {highest_month} ({highest_temp}°C)")

# 2. Calculate average temperature
# Extract all temperatures into a list
temp_values = [temp for month, temp in temperatures]
average_temp = sum(temp_values) / len(temp_values)
print(f"2. Average temperature: {average_temp:.2f}°C")

# 3. Create a new list with temperatures converted to Fahrenheit
# Formula: F = C * 9/5 + 32
temp_fahrenheit = [(month, temp * 9/5 + 32) for month, temp in temperatures]
print("3. Temperatures in Fahrenheit:")
for month, f in temp_fahrenheit:
    print(f"   {month}: {f:.1f}°F")

# 4. Find months with temperature above average
above_avg_months = [month for month, temp in temperatures if temp > average_temp]
print(f"4. Months with temperature above average ({average_temp:.2f}°C): {above_avg_months}")
```

```
1. Month with highest temperature: May (18°C)
2. Average temperature: 10.60°C
3. Temperatures in Fahrenheit:
   Jan: 41.0°F
   Feb: 42.8°F
   Mar: 50.0°F
   Apr: 57.2°F
   May: 64.4°F
4. Months with temperature above average (10.60°C): ['Apr', 'May']
```

Exercise 8

```python
# Create a tuple of tuples: (month, days)
months_days = (
    ("January", 31),
    ("February", 28),
    ("March", 31),
    ("April", 30),
    ("May", 31),
    ("June", 30),
    ("July", 31),
    ("August", 31),
    ("September", 30),
    ("October", 31),
```

```
    ("November", 30),
    ("December", 31)
)

# 1. Print all months with 31 days
print("Months with 31 days:")
for month, days in months_days:
    if days == 31:
        print(month)

# 2. Ask user for a month and show how many days it has
user_month = input("\nEnter a month name: ")

found = False
for month, days in months_days:
    if month == user_month:
        print(f"{month} has {days} days.")
        found = True
        break

if not found:
    print("Month not found. Please check the spelling.")
```



Exercise Solutions in chapter 7

Exercise 1

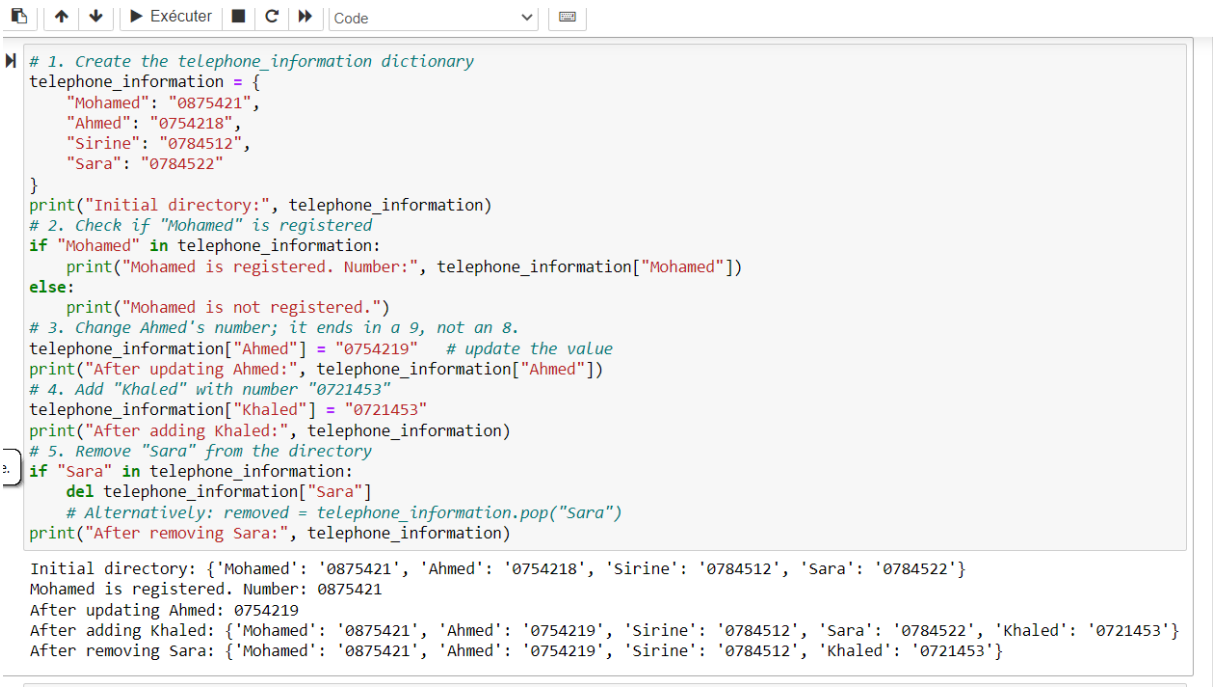

```
# Create an inventory dictionary (product: quantity)
inventaire = {
    'pommes': 100,
    'bananes': 80,
    'poires': 120
}

# Display the dictionary
print("Inventory dictionary:", inventaire)

# Use values() to get all quantities
quantities = inventaire.values()
print("Quantities (values):", list(quantities))   # convert to list for display

# Use keys() to get all product names
products = inventaire.keys()
print("Products (keys):", list(products))

# Use items() to get key-value pairs and iterate
print("\nIterating with items():")
for key, value in inventaire.items():
    print(f"{key}: {value}")
```

```
Inventory dictionary: {'pommes': 100, 'bananes': 80, 'poires': 120}
Quantities (values): [100, 80, 120]
Products (keys): ['pommes', 'bananes', 'poires']

Iterating with items():
pommes: 100
bananes: 80
poires: 120
```

Exercise 2

```
# 1. Create the telephone_information dictionary
telephone_information = {
   "Mohamed": "0875421",
   "Ahmed": "0754218",
   "Sirine": "0784512",
   "Sara": "0784522"
}

print("Initial directory:", telephone_information)

# 2. Check if "Mohamed" is registered
if "Mohamed" in telephone_information:
   print("Mohamed is registered. Number:", telephone_information["Mohamed"])
```

```
else:
    print("Mohamed is not registered.")

# 3. Change Ahmed's number; it ends in a 9, not an 8.
telephone_information["Ahmed"] = "0754219"   # update the value
print("After updating Ahmed:", telephone_information["Ahmed"])

# 4. Add "Khaled" with number "0721453"
telephone_information["Khaled"] = "0721453"
print("After adding Khaled:", telephone_information)

# 5. Remove "Sara" from the directory
if "Sara" in telephone_information:
    del telephone_information["Sara"]
    # Alternatively: removed = telephone_information.pop("Sara")
print("After removing Sara:", telephone_information)
```



Exercise Solutions in chapter 9

Exercise 1

```
# Step 1: Create a new text file named hello.txt and write "Hello, World!" into it.
with open("hello.txt", "w") as file:
    file.write("Hello, World!")

# Step 2: Open the file and print its contents to the console.
print("Initial content:")
```

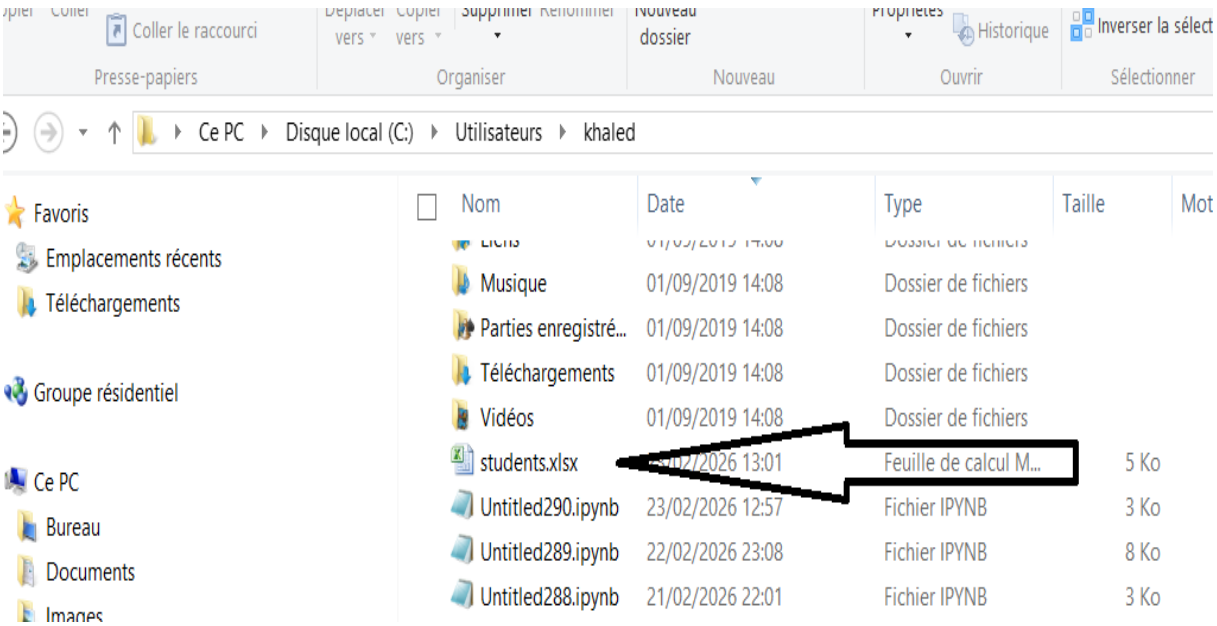
```
with open("hello.txt", "r") as file:
    content = file.read()
    print(content)

# Step 3: Append a new line to the file, then read and display the updated file.
with open("hello.txt", "a") as file:
    file.write("\nThis is a new line.")

print("\nUpdated content after appending:")
with open("hello.txt", "r") as file:
    updated_content = file.read()
    print(updated_content)

# Step 4: Count the total number of words in the file.
with open("hello.txt", "r") as file:
    text = file.read()
    words = text.split()
    word_count = len(words)

print(f"\nTotal number of words in the file: {word_count}")
```

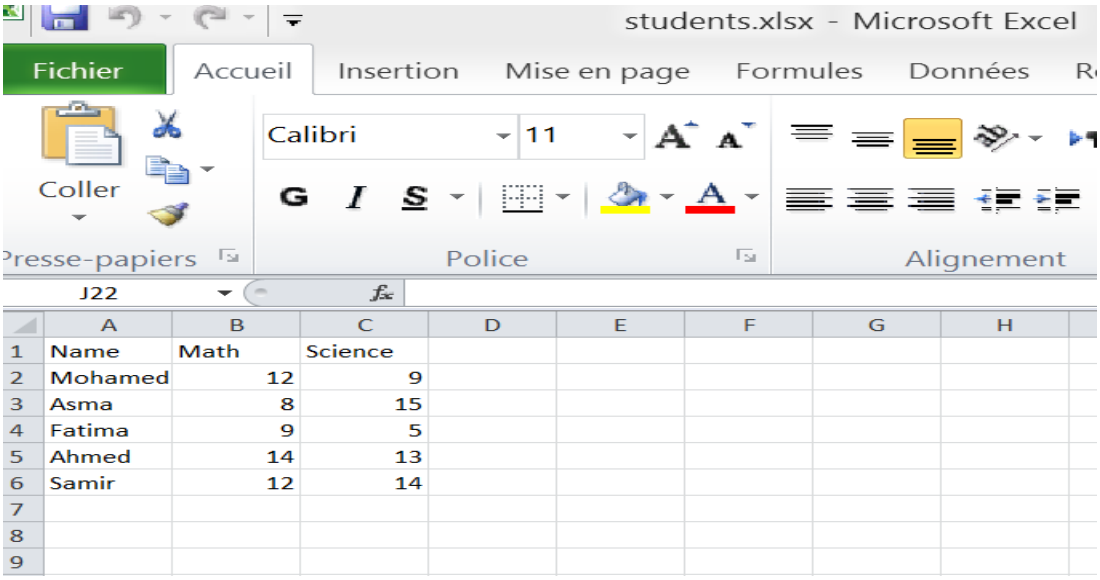
Exercise 2

```
import openpyxl
from openpyxl import Workbook, load_workbook
# 1. Create a new Excel file named students.xlsx
workbook = Workbook()
sheet = workbook.active
sheet.title = "Scores"

# 2. Add headers
sheet["A1"] = "Name"
sheet["B1"] = "Math"
sheet["C1"] = "Science"

# 3. Write at least 5 rows of data
data = [
    ("Mohamed", 12, 9),
    ("Asma", 8, 15),
    ("Fatima", 9, 5),
    ("Ahmed", 14, 13),
    ("Samir", 12, 14)
]

for row_idx, (name, math, science) in enumerate(data, start=2):  # start from row 2
    sheet.cell(row=row_idx, column=1, value=name)
    sheet.cell(row=row_idx, column=2, value=math)
    sheet.cell(row=row_idx, column=3, value=science)
```

```
# Save the file
workbook.save("students.xlsx")

# 4. Re-open the same file, read all rows, calculate averages, and print
workbook = load_workbook("students.xlsx")
sheet = workbook["Scores"]  # or workbook.active

print("Student Averages:")
for row in sheet.iter_rows(min_row=2, values_only=True):  # skip header
    name, math, science = row
    if name is None:  # in case of empty rows
        continue
    average = (math + science) / 2
    print(f"{name}: {average:.2f}")
```

Once the program is executed, it will automatically create students.xlsx and then automatically open and display the students' averages.





	A	B	C	D	E	F	G	H
1	Name	Math	Science					
2	Mohamed	12	9					
3	Asma	8	15					
4	Fatima	9	5					
5	Ahmed	14	13					
6	Samir	12	14					
7								
8								
9								

```python
import openpyxl
from openpyxl import Workbook, load_workbook

# 1. Create a new Excel file named students.xlsx
workbook = Workbook()
sheet = workbook.active
sheet.title = "Scores"

# 2. Add headers
sheet["A1"] = "Name"
sheet["B1"] = "Math"
sheet["C1"] = "Science"

# 3. Write at least 5 rows of data
data = [
    ("Mohamed", 12, 9),
    ("Asma", 8, 15),
    ("Fatima", 9, 5),
    ("Ahmed", 14, 13),
    ("Samir", 12, 14)
]

for row_idx, (name, math, science) in enumerate(data, start=2):  # start from row 2
    sheet.cell(row=row_idx, column=1, value=name)
    sheet.cell(row=row_idx, column=2, value=math)
    sheet.cell(row=row_idx, column=3, value=science)

# Save the file
workbook.save("students.xlsx")

# 4. Re-open the same file, read all rows, calculate averages, and print
workbook = load_workbook("students.xlsx")
sheet = workbook["Scores"]  # or workbook.active

print("Student Averages:")
for row in sheet.iter_rows(min_row=2, values_only=True):  # skip header
    name, math, science = row
    if name is None:  # in case of empty rows
        continue
    average = (math + science) / 2
    print(f"{name}: {average:.2f}")
```

```
Student Averages:
Mohamed: 10.50
Asma: 11.50
Fatima: 7.00
Ahmed: 13.50
Samir: 13.00
```

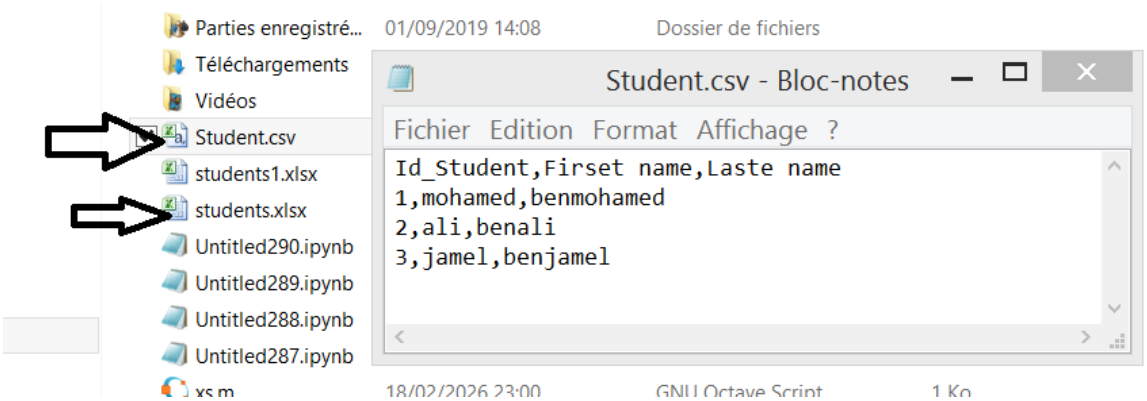
automatically open and display the students' averages.

Exercise 3

- The csv module is part of Python's standard library.
- The openpyxl library is required for Excel operations. Install it with: pip install openpyxl

```
import csv
import openpyxl
from openpyxl import Workbook

# Student data as a list of dictionaries
students = [
    {"Id_Student": 1, "Firset name": "mohamed", "Laste name": "benmohamed"},
    {"Id_Student": 2, "Firset name": "ali", "Laste name": "benali"},
    {"Id_Student": 3, "Firset name": "jamel", "Laste name": "benjamel"},
]
# Define the column headers (in the desired order)
fieldnames = ["Id_Student", "Firset name", "Laste name"]
# 1. Save to CSV file
with open("Student.csv", mode="w", newline="", encoding="utf-8") as csv_file:
    writer = csv.DictWriter(csv_file, fieldnames=fieldnames)
    writer.writeheader()        # Write the header row
    writer.writerows(students)    # Write all student rows

print("CSV file 'Student.csv' created successfully.")
# 2. Save to Excel file
workbook = Workbook()
sheet = workbook.active
sheet.title = "Students"
# Write the header row
sheet.append(fieldnames)
# Write each student's data
for student in students:
    sheet.append([student["Id_Student"], student["Firset name"], student["Laste name"]])
workbook.save("students.xlsx")
print("Excel file 'students.xlsx' created successfully.")
```

Once the program is executed, it will automatically create : CSV file 'Student.csv' created successfully. Excel file 'students.xlsx' created successfully.



Bibliographical references:

[1] . Allen B. Downey Think Python: How to Think Like a Computer Scientist, O'Reilly Media, 2015;

[2] . Zed A. Shaw Learn Python 3 the Hard Way: A Very Simple Introduction to the Terrifyingly Beautiful World of Computers and Code, Addison-Wesley Professional, 2017;

[3] . Barry, P. Head first Python: A brain-friendly guide. " O'Reilly Media, Inc.", 2016;

[4] . Ramalho, L.. Fluent Python. " O'Reilly Media, Inc.", 2022;

[5] . Swinnen, G.. Apprendre à programmer avec Python 3. Editions Eyrolles, 2012;

[6] . Le Goff, V.. Apprenez à programmer en Python. Editions Eyrolles, 2019;

[7] . Matthes, E. Python crash course: A hands-on, project-based introduction to programming. no starch press, 2019;

https://www.python.org/downloads/

https://www.anaconda.com/download

https://docs.python.org/3/index.html

https://www.w3schools.com/python/

https://python-course.eu/python-tutorial/