REPUBLIC OF SENEGAL

*_*_*_*_*_*_*

MMINISTRY OF HIGHER EDUCATION, RESEARCH AND INNOVATION

*_*_*_*_*_*_*

AFRICAN INSTITUTE FOR MATHEMATICAL SCIENCES

*_*_*_*_*_*_*

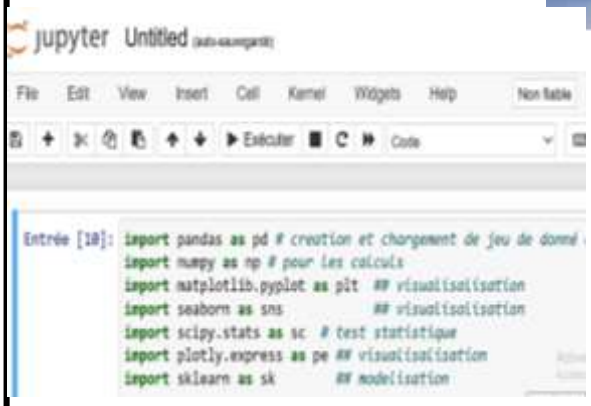
DEPARTMENT OF MATHEMATICS

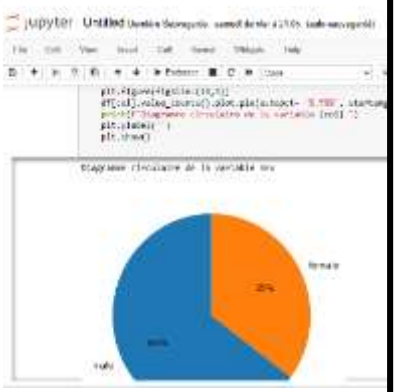
*_*_*_*_*_*_*

MASTER RESEARCH IN MATHEMATICS AND APPLICATIONS

6th Edition

PROGRAMMAMING WITH PYTHON





Dr. Franck YOVO
(Assistant Professor/CAMES FAST_NATITINGOU /UNSTIM_BENIN)

Academic year :
2023 - 2024

Table of contents

Foreword

A few words about the origin of this course

This course was originally designed for students beginning in Python programming in the Bachelor of Chemistry, Physics and Mathematics courses of the Faculty of Science and Technology of the National

University of Science, Technology, Engineering and Mathematics; from the Institute of Mathematics and Physical Sciences of the University of Abomey and more especially for students of the Research Master in Mathematics and Applications of the African Institute for Mathematical Sciences.

This course is based on version 3 of Python, the version most used by the scientific community. If you notice any errors when reading this document, please report them to us. The course is available in PDF 3 and Word versions.

Acknowledgment

We would like to thank Dr Coura Balde and the entire coordination team for allowing us to deliver programming lessons with Python at the African Institute for Mathematical Sciences. A big thank to Dr. Yaé U. Gaba, Research Professor at the Institute of Mathematics and Physical Sciences of the University of Abomey Calavi for authorizing us to use the modules from his GitHub site. We thank Dr. Fabrice Elegbede and Dr Amoussatou Sakirigui, Research Professor at the Faculty of Science and Technology of Natitingou of the National University of Science, Technology, Engineering and Mathematics for the modules and tutorials.

Thank also to all the contributors, occasional or regular: Rominus Segla, Charif Moumouni and Bio Orou Bouyagui.

Finally, thank to the tutors: Balbine O. Etoga Mboua, Anoumou Attiogbe, Martial A. Ngounou and Yaulande Douanla for supporting us in the success of this module.

Main objective

Give students the necessary tools to write programs in Python for the digital resolution of scientific problems.

Specific objectives

At the end of the learning activities, the student must be able to:

- understand python basic data types (int, long, float, bool)
- understand and manipulate text data (string) and their methods
- understand and use python's modules
- write python scripts to solve easy to fairly difficult math problems
- define and manipulate functions
- understand and use lists, tuples and dictionaries and their methods
- manipulate arrays and array's methods
- understand and use random numbers
- plot 2D graphs using matplotlib
- write a program to estimate and plot the solution of a system of ODEs
- write a program to estimate the value of a single, double or triple integral
- develop a computer program given a numerical algorithm

- analyze and interpret the outputs of a computer program
- write a short report explaining the program design and results

Prerequisites: This course requires basic notions of mathematics, science and computer science.

Keywords

- Computer architecture
-computer science
- Python language instruction
- Algorithm
-coded
- Functions and variables

Evaluation methods

- Continuous checks
- Final exam: 60%
- Presentation and practical work: 40%

Bibliography

1- Gaba U. Yaé. Bases_de_programmation_python-cahiers-virtuels-de-cours, https://github.com/gabayae/bases_de_programmation_python-cahiers-virtuels-de-cours/blob/main/00.Introduction_à_python.ipynb

2- KPOTIN G. Cours de Chimie numérique ; Licence Chimie Fondamentale ; UAC 2018.45p.

3- Tarek Z. Programmation Python, éditions Eyrolles, Paris, 2009, 586 p.

4- Wesley J ; Baland M.C ;Bohy A ; Carite L. Au coeur de Python, editions CampusPress, Paris, 2007, 385 p.

5- Swinnen G. Apprendre à programmer python 3, edition Eyrolles –Paris, 473p.

6- Cordeau R . Cours sur Python 3,IUT d'Orsay, ttp://www.afpy.org/Members/bcordeau/Python3v1-1.pdf/download

7- Lycée Faidherbe. TP INFORMATIQUE, 2019-2020

8- Therincourt David. Python pour la physique chimie, 40p

9- . Charles R. Severance (2016) Python for everybody: Exploring Data Using Python 3. Sue Blumenberg, Elliott Hauser, ISBN 1530051126.

10- 2. Mark Summerfield (2010) Programming in Python 3: A Complete Introduction to the Python Language, 2nd Edition. Addison Wesley

11- 3. Mark Lutz and David Ascher (2007) Learning Python, 3rd Edition. O'Reilly Media

12- 4. Peter Norton, Alex Samuel, David Aitel, Eric Foster-Johnson, Leonard Richardson, Jason Diamond, Aleatha Parker, Michael Roberts (2005) Beginning Python. Wrox

13- 5. Hans Petter Langtangen (2008), Python scripting for Computational Science (3rd edition), Texts in Computational Science and Engineering, Springer-Verlag Berlin Heidelberg

Course Websites

1. python.aims.ac.za: For online lecture notes
2. https://github.com/gabayae/bases_de_programmation_python-cahiers-virtuels-de-cours/blob/main/00.Introduction_à_python.ipynb
3. https://pythonnumericalmethods.berkeley.edu/notebooks/chapter22.06-Python-ODE-Solvers.html

Introduction

Computer programming, sometimes also computational, is a branch of science that uses mathematical laws and scientific notions to calculate the structures and properties of objects such as molecules, solids, aggregates (or clusters), surfaces, etc. The boundary between the simulation carried out and the real system is defined by the level of precision required and/or the complexity of the systems studied and the theories used during modeling. The properties sought may be structure and statistical analyses. The advantages of Python are numerous. On the one hand, it is undoubtedly the most used scripting language in the world because you can do almost everything with it: web programming, statistics, machine learning, database management. The Python community is the largest in the programming world.

Applications

In programming, chemists, physicists and mathematicians develop algorithms and codes in order to predict properties, molecular or otherwise, and possibly chemical reaction paths. Numerical scientists, on the other hand, can simply apply existing codes and methodologies for specific problems.

Chapter 0. Reminder on the use of IT tools

Before diving into the exploration of programming and its application in digital chemistry, it is essential to understand the functioning of the essential tool for computer programming and automation in general: the computer. What is a computer? What are its essential components? How to select the right computer? So many questions that the following sub-chapter will attempt to answer.

0.1. Computer architecture

A computer is an electronic machine designed to perform various logical and arithmetic operations by processing data according to previously defined instructions. In other words, it is a programmable device capable of manipulating information and performing tasks by following sequences of instructions. Modern computers are based on the Von Neumann architectural model, which defines the fundamental components and key interactions of a computer. Here are the main elements that make up a computer:

• Central processing unit (CPU): This is the brain of the computer. The CPU executes the instructions and performs the calculations. It is made up of the control unit (which coordinates operations) and the arithmetic and logic unit (which carries out mathematical and logical operations).

• Memory: Memory is used to store data and instructions. It is generally divided into random access memory (RAM) for data in use and storage memory (hard drives, SSD) for long-term data.

• Input/output devices: These are the devices that allow the computer to interact with the outside world. Examples of peripherals are keyboards, mice, displays, printers, speakers, etc.

• Data Bus: Data buses are the paths by which data flows between computer components. There are buses for transmitting data, addresses and control.

• Software: Software consists of the programs and instructions that direct the operation of the computer. Operating systems, applications and basic programs are part of the software.

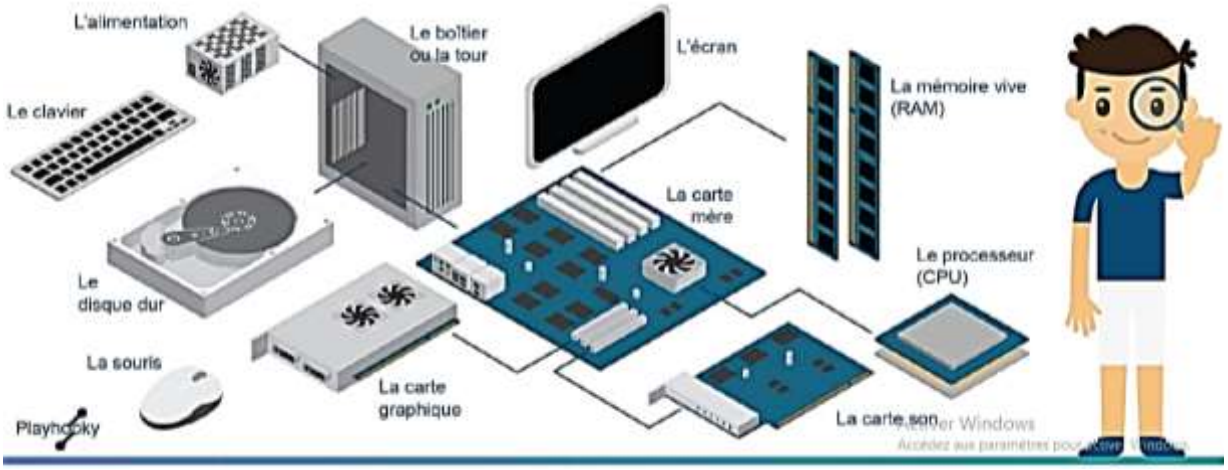


Figure N°1: The building blocks of a computer

You will find additional information on the function and usefulness of each component by clicking on this link.

Operating systems

An operating system is fundamental software that manages a computer's hardware and software resources. It provides an interface between the user, applications and computer hardware. Operating systems are designed to handle multiple tasks and processes simultaneously, optimizing resource usage and providing a user-friendly environment.

Main roles of operating systems

Operating systems allocate and control hardware resources (CPU, memory, disks, etc.) between different running applications and processes. This ensures efficient use of available resources.

0.2. Algorithms and data structure[1]

An algorithm is a sequence of precise steps or instructions for solving a given problem. It is a series of logical and orderly actions that, when followed correctly, leads to the desired solution. Algorithms are essential in computer science for solving problems efficiently and reproducibly.

[1] https://youtu.be/kk6YbA5I-Iw?list=PL2aehqZh72Lumvy4tSekr6Rzcgwn15MLI

0.2.1. Key elements of an algorithm

Inputs: The data provided to the algorithm to carry out its processing.

This data can be of different types, such as numbers, strings, lists, etc.

Processing: The specific steps that the algorithm follows to transform the input data to output data. This often involves mathematical operations, comparisons, loops, etc.

Exits : The results produced by the algorithm after processing the input data according to the specified steps.

To design an algorithm to solve a given scientific problem, several steps are necessary. First of all, it is imperative to:

• Understand the Problem: Before starting to write the algorithm, it is essential to clearly understand the problem to be solved. This involves identifying the required input data as well as the expected results.

• Define Steps: Breaking the problem into smaller, more manageable steps is crucial. Each step should represent a specific logical or arithmetic operation to be performed.

• Use Control Structures: Control structures such as loops (to repeat actions) and conditional statements (to make decisions based on conditions) are essential tools to guide the flow of the algorithm.

• Write Pseudo-code: Developing pseudo-code for the algorithm is recommended. Pseudo-code constitutes an informal language describing the steps of the algorithm without worrying about the syntax specific to a particular programming language.

By following these steps, you will be able to design an efficient and accurate algorithm to solve the scientific problem at hand.

Example:

Algorithm: Calculation of the Average

Input: Two numbers (n1, n2) Output: The average of the two numbers

1 Read n1 from input

2 Read n2 from input

3 Calculate the sum of n1 and n2, and store in sum

4 Calculate the average by dividing the sum by 2, and store in average

5 Show average value as result

Control structure: Conditional statements

Conditional statements allow an algorithm to make decisions based on conditions. The classic example is the if...else statement.

1. If x > 0 Show “x is positive”

2. Otherwise Show "x is negative or zero"

3. End If

Loops: Loops allow an algorithm to repeat a series of instructions. The most common loops are for and while loops.

1. For i ranging from 1 to 5 Show i

2. End For

Sorting Algorithm

Sorting algorithms are used to organize data in a specific order. Among the best known are bubble sort, selection sort and quick sort (QuickSort).

Algorithm SortBySelection(list)

1. For i going from 0 to size(list) - 1 minIndex = i

2. For j going from i + 1 to size(list)

3. If list[j] < list[minIndex] minIndex = j

4. End If

5. End To Exchange list[i] with list[minIndex]

6. End For End

Recursion

Recursion is a concept where a function calls itself to solve a problem. Recursive algorithms are used to solve problems that can be broken down into similar subproblems.

Factorial(n) function

1. If n <= 1 Return 1

2. Else Return n * Factorial(n - 1)

3. End If End Function

Dynamic Programming

Dynamic programming consists of solving problems by breaking down their resolution into sub-problems, memorizing intermediate results to avoid recalculation.

Fibonacci sequence algorithm

Fibonacci function(n, memo)

1. If n in memo Return memo[n]

2. If n <= 1 Return n

3. Otherwise result = Fibonacci(n - 1, memo) + Fibonacci(n - 2, memo)

4. memo[n] = result

5. Return result

6. End If End Function

This represents a deeper insight into algorithmics. Each example illustrates different concepts and approaches to solving problems using algorithms.

Examples of algorithm applications in chemistry

Chemical Reaction Simulation

Problem: You want to simulate chemical reactions and calculate the quantities of reactants and products

Example: Simulation of the combustion reaction of methane (CH4) with

oxygen (O2) to form carbon dioxide (CO2) and water (H2O).

Equation: CH4 + 2O2 -> CO2 + 2H2O

Algorithm: Reaction Simulation

1. Input: Equation, QuantityCH4, QuantityO2, CoefficientCH4, CoefficientO2 Output: Equation, molCO2 formed, molH2O formed
2. Read Equation from input
3. Read QuantityCH4, from input
4. Read QuantityO2 from input
5. Read CoefficientCH4 from input
6. Read CoefficientO2 from input
7. Read CoefficientCO2 from the input
8. Read CoefficientH2O from input
9. Calculate ratio between QuantityCH4 and CH4 coefficient, and store in ratio1
10. Calculate ratio between QuantityO2 and coefficient O2, and store in ratio2
11. Calculate molCO2=ratio1* CoefficientCO2 and store in mol1CO2
12. Calculate molCO2=ratio2* CoefficientCO2 and store in mol2CO2
13. If ratio1 < ratio2 Return mol1CO2
14. Otherwise Return mol1CO2
15. End If End

0.3. Getting started with the Linux operating system[2]

Linux is a cheaper, more secure, more portable operating system (than commercial alternatives). It is widely used by servers, network equipment, Android devices and other “small” devices (e-readers, raspberry pi, etc.). Linux is actually the name of the operating system (OS) kernel.[3]

[2]https://youtu.be/IVquJh3DXUA

A complete OS based on the Linux kernel is called a distribution (or distro). Example: Debian, Ubuntu (based on Debian), Fedora, Arch Linux, Slackware etc…The use of the different command lines used on Linux are detailed through the links below[4].

Figure N°2: Ubuntu desktop environment

1 Introduction to Python

1.1 What is Python?

1.1.1 Brief presentation of Python and its uses.

Python is a powerful, versatile and popular programming language that has gained notoriety over the years thanks to its simplicity, readability and versatility. Designed by Guido van Rossum and first released in 1991, Python is now widely used around the world for a multitude of applications, from website building to data science, through task automation and system programming. One of the most notable features of Python is its clear and readable syntax. The language was designed to be easy to learn and understand, making it accessible to beginners yet powerful for experienced programmers. Indentation (spacing lines) is an integral part of Python's syntax, which promotes code readability. Python is a versatile language that can be used for a variety of tasks. You can create web applications, develop games, perform complex data analysis, automate repetitive tasks, and much more. Its extensive standard library offers modules for a wide variety of domains, which means many features are already ready to be used. Python benefits from an active and engaged developer community. This means that you have access to numerous resources, tutorials, third-party libraries and discussion forums for your help in your Python learning journey. For this course, we will use Python on Jupiter environment.

Jupyter Notebook is an open source web application that lets you create and share interactive code, visualizations, and more. This tool can be used with several programming languages, including Python.

[4] https://perso.liris.cnrs.fr/pierre-antoine.champin/enseignement/linux/s1.html.

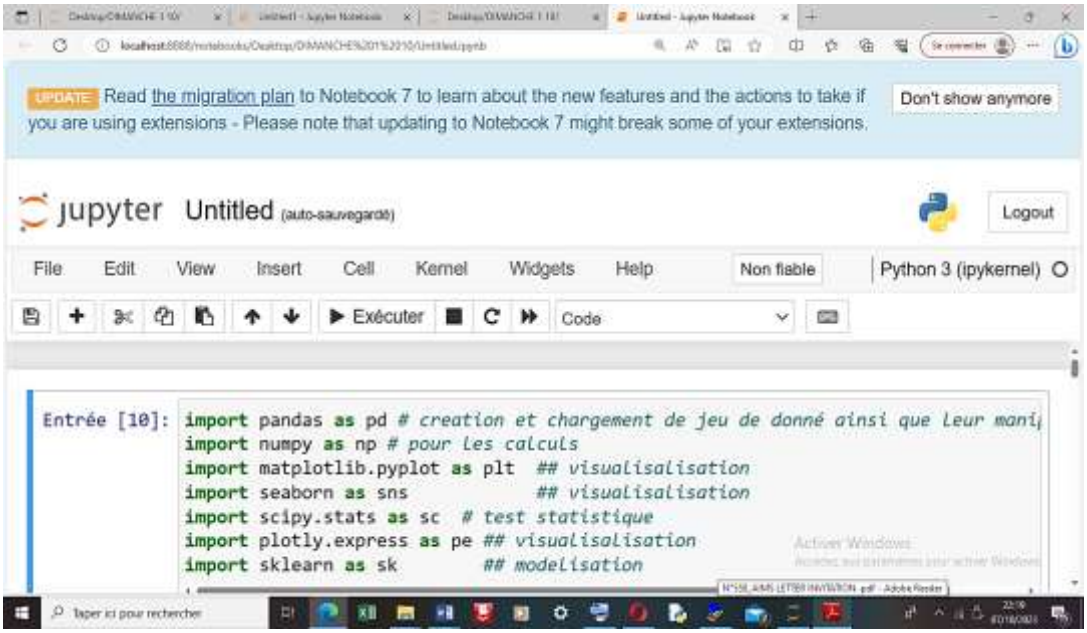


Figure N°3: Jupiter Notebook

1.1.2 Benefits of learning Python.

Python is widely used in industry and academia. Companies such as Google, Facebook, Instagram, Dropbox, and many others use Python to develop software and services. In the academic field, Python has become an essential tool for scientific research and data analysis. There are many benefits to learning Python. Here are some of the most significant:

• Ease of Learning: Python is known for its ease of learning, which makes it makes an ideal choice for those just learning about programming. Its simple syntax and readable allows you to focus on statistical concepts rather than programming in herself

• Data Analysis: Python has become an essential tool for data analysis in statistics. Libraries such as NumPy, Pandas and SciPy provide functionality powerful tools for data manipulation, cleaning and analysis, allowing you to work efficiently on complex data sets.

• Data Visualization: The Matplotlib library allows you to create graphs and custom data visualizations. This is an essential asset for understanding and visually present your statistical results.

• Advanced Statistical Analysis: Python offers great flexibility for performing advanced statistical analyses. You can use specialized libraries like Statsmodels and scikit-learn to perform regressions, hypothesis testing, learning machine, and much more.

• Reproducibility: Python encourages reproducibility of research. You can create Jupyter scripts and notebooks to document and reproduce your analyses, which is essential in the academic world.

• Integration with Other Tools: Python integrates easily with other tools commonly used academic tools, such as LaTeX, R, and databases, making workflow easier in your research projects.

• Active Community: Python benefits from a constantly growing community, which means you can find online help, tutorials and discussion forums to answer your questions and resolve specific statistical problems.

• Promising Careers: By acquiring skills in Python, you strengthen your attractiveness on the job market. Many career opportunities in statistics now require Python programming skills.

• Abundant Educational Resources: There are many educational resources free online courses, including courses, books and tutorials, that help you develop your Python skills at your own pace. Learning Python is a valuable step in your academic journey. Not only it makes data manipulation and analysis easier, but it also strengthens your skills and your career prospects in several scientific fields. Whether you plan to work in research, industry or any other field, Python will be a major asset for your studies and your professional future.

1.2 Installation and configuration

Installing and configuring Python are essential steps to start programming with this language. Here is a step-by-step guide to help you install and configure Python on your system:

1.2.1 Step 1: Downloading Python

1. Go to the official Python website at https://www.python.org/downloads/ to download the latest version of Python.

2. Choose the version of Python that matches your operating system. If you use Windows, you can opt for the Windows version to install (e.g. Python 3.x.x).

If you're using macOS or Linux, Python usually comes preinstalled, but you can Also choose to update it to the latest version.

3. Click the download link to start downloading the installation file.

1.2.2 Step 2: Installing Python

1. Once the installation file is downloaded, open it.

2. Check the “Add Python X.X to PATH” box during installation (replace X.X with the version of Python that you downloaded). This option adds Python to your path system access, making it easy to run Python from anywhere on your computer.

3. Click “Install Now” to start the installation. The installer will copy the necessary files on your system.

4. Wait for the installation to complete. Once completed, you will see a message saying that Python has been installed successfully.

1.2.3 Step 3: Verification of the Installation

To verify that Python was installed correctly, open your command prompt (on Windows) or terminal (on macOS and Linux), then type the following command:

python --version

You should see the version of Python you have installed.

1.2.4 Step 4: Installation of a Development Environment

Although you can program in Python using just a text editor, it is recommended to install an integrated development environment (IDE) for a smoother experience. more user-friendly development. Two popular IDEs for Python are PyCharm and Visual Studio Code, but there are many others.

In this course, we will mainly use Anaconda, a platform which brings together several development environments and essential Python libraries. To run the programs in this course on your own computer, you will need to have Python, as well as the NumPy and Matplotlib libraries installed. A practical solution for quickly configuring The development environment required for this course is to install Anaconda, a free software. Anaconda offers you a simple and complete installation of everything you need: https://www.anaconda.com/download/

We recommend using Python version 3.6 for this course, and Anaconda includes a integrated development environment called Spyder (Scientific PYthon Development EnviRonment) which will be particularly useful to you.

1.3 Getting Started with Python

1.3.1 Using the Python interpreter to perform simple calculations.

The Python interpreter is a powerful tool that allows you to perform simple calculations as well than more complex tasks using the Python programming language.

1. Open the Python interpreter: To open the Python interpreter on your system, open a command prompt (on Windows) or terminal (on macOS or Linux) and just type “python” followed by the “Enter” key. You will then see an interactive prompt Python which looks like this:

```
Python 3.9.6 (default, Jun
9 2021, 10:58:55) [GCC
8.4.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

2. Perform calculations: Now you can use the Python interpreter as a calculator. For example, to perform an addition, simply type the operation math and press “Enter”. For example, to add 3 and 5, type:

```
>>> 3 + 5
```

The interpreter will return the result:

8

You can perform basic math operations such as addition (+), subtraction (-), multiplication (*), division (/), and other more advanced operations in using built-in math functions.

3. Using Variables: You can also use variables to store values and perform calculations with them. For example, to set a variable a equal to 10 and a variable b equal to 20, type:

```
>>> a = 10
>>> b = 20
```
Then you can perform calculations with these variables:

```
>>> c = a + b

>>> c
```

The interpreter will return the result of the addition in the variable c.

4. Math functions: Python also has many math functions- integrated technologies. For example, to calculate the square root of a number, you can use the sqrt function of the math module:

```
>>> import math
>>> math.sqrt(25)
```
This will return the value of the square root of 25.

The Python interpreter is a great way to perform simple calculations and test quickly ideas using the Python language. It also offers the possibility of carrying out operations more complex by writing complete Python programs. To exit the Python interpreter, you can simply type exit() or quit() and press “Enter”.

1.3.2 Introduction to variables and data types (int, float, str, bool).

Variables and data types are fundamental concepts in programming. They allow you to store, manipulate and organize information in a program. Here is a introduction to variables and data types in Python, one of the programming languages the most popular.

Variables:

A variable is a name you give to a memory location to store data. This allows you to reuse and manipulate this data in your program. In Python, you You can declare a variable by assigning it a value. For example :

```
x = 10
y = "Hello"
```
In this example, x is an integer variable (int) that stores the value 10, and y is a variable of type string (str) which stores the text “Hello”. You can use names of variable descriptions to make your code more readable.

Common data types in Python:

1. int (integer): Integer variables store integer numbers, positive or negative, without decimals. For example :

```
age = 30
```

2. float: Float variables store floating point numbers, that is, real numbers with a decimal part. For example :

```
price = 19.99
```

3. str (string): String variables store text. You can use single quotes (’ ’) or double quotes (” “) to define a string of characters. For example :

```
name = "Alice"

message = 'Hello, how are you?'
```

4. bool (boolean): Boolean type variables can only have two values: True (true) or False. They are often used to represent states or conditions. For example :

```
is_connected = True

is_major = False
```

Using variables:

You can perform operations on variables using appropriate operators by depending on their type. For example, you can perform mathematical calculations on integers or floats, concatenate character strings, or evaluate Boolean expressions.

Example :

```
a = 5

b = 3

sum = a + b # Addition of two integers

name = "Alice"

message = "Hello, " + name + "!" # Concatenation of strings

is_major = (age >= 18) # Evaluation of a Boolean expression
```

Variables and data types are essential in programming to store, manipulate and organize information. Python offers great flexibility in terms of data types, this which facilitates the creation of complex programs. It is important to choose the data type appropriate depending on what you want to accomplish in your program.

1.3.3 Assigning values to variables.

Assigning values to variables is the action of giving a value to a variable so that it can be used in a program. In Python you can assign values to variables as follows : variable_name = value

Here are some examples of assigning values to variables for different data types:

1. Assignment of an integer (int):

   age = 30

   In this example, the variable age is declared and given the value 30. It stores an integer.

2. Assignment of a float:

   price = 19.99

   The price variable is declared and given the value 19.99. It stores a decimal number floating.

3. Assignment of a character string (str):

   name = "Alice"

   The variable name is declared and contains the character string “Alice”.

4. Assignment of a boolean (bool):

   is_connected = True

   The variable is_connected is declared and it is set to True, which means that the user is logged in.

5. Assigning multiple values: You can also assign values to multiple variables in a single statement:

   x, y, z = 1, 2, 3

   Here, the variables x, y and z are given the values 1, 2 and 3 respectively.

6. Dynamic assignment: In Python, variables are dynamically typed, which means you don't need to explicitly specify their type. The type of the variable is determined automatically depending on the value you assign to it. For example :

   my_variable = 42 # my_variable is an integer (int)

   my_variable = "Hello" # my_variable is now a string (str)

   You can change the type of a variable by assigning it a new value of a type different.

Once you assign a value to a variable, you can use it in expressions, calculations or conditional statements to perform operations based on this value. Variables allow you to store and manipulate data flexibly in your Python programs.

1.3.4 Displaying results with the print() function.

The print() function in Python is used to display information to the console or in the standard output. You can use it to display variable values, text messages, calculation results, etc.

print(expression)

where expression is what you want to display. Here are some examples :

1. **Show text:**

   print("Hello, how are you?")

   This instruction will display the message “Hello, how are you?” at the Consol.

2. **Display variable values:**

   You can display variable values by passing them as arguments to print():

   age = 30

   print("Age:", age)

   This will display “Age: 30” on the console.

   3. Display calculation results:

[2]:
```
x = 5
y = 10
sum = x + y
print("The sum of", x, "and", y, "is equal to", sum)
```

The sum of 5 and 10 is equal to 15.

This will display “The sum of 5 and 10 equals 15” to the console.

4. Display multiple elements with print():

You can display multiple elements by separating them with commas in print(). Python will automatically add a space between displayed elements:

[3]: first name = "Alice"

age = 25

print("First name:", first name, "Age:", age)

First name: Alice Age: 25

This will display “First Name: Alice Age: 25” to the console.

5. Display formatted values:

You can also format the display using f-strings (available from Python 3.6 and later) or the format() method. For example :

[4]: name = "Smith"

first name = “John”

age = 35 print(f"Last name: {last name}, First name: {first name}, Age: {age}")

Name: Smith, First name: John, Age: 35

This will display “Last Name: Smith, First Name: John, Age: 35” on the console. The print() function is useful for obtaining information about the execution of your program, debug problems, display results to the user, or simply to understand what is happening inside your code. It is widely used in Python development.

1.4 Application and discovery exercises

Exercise 1: Calculating the average

Write a Python program that calculates the average of three integers (for example, 10, 15, and 20) and displays the result.

Exercise 2: Temperature conversion

Write a program that asks the user to enter a temperature in degrees Celsius, then convert this temperature to degrees Fahrenheit using the formula: Fahrenheit = (Celsius * 9/5) + 32

Then display the temperature in degrees Fahrenheit.

Exercise 3: Concatenating character strings

Ask the user to enter their first and last name, then display a greeting message personalized, for example: “Hello, [first name] [last name]!”

Exercise 4: Calculating the perimeter and area of a rectangle

Ask the user to enter the length and width of a rectangle. Then calculate and display the perimeter and area of the rectangle using the following formulas:

• Perimeter = 2 * (length + width)

• Area = length * width

Exercise 5: Currency conversion

Write a program that asks the user to enter a certain amount in US dollars. Convert this amount into euros using an exchange rate that you define (for example example, 1 dollar = 0.85 euros). Show the amount in euros.

Exercise 6: Mathematical operations

Write a program that does the following:

• Ask the user to enter two integers.

• Calculate and display the sum, difference, product and quotient of these two numbers.

These exercises should help you consolidate your knowledge of basic Python concepts that we have covered so far. Do not hesitate to solve them and experiment for better understand how variables, data types, and the print() function work in Python.

2 Control Structures

In Python, control structures are used to manage the execution flow of a program. They make it possible to make decisions, iterate on data sequences and control the code execution flow. Here are the main control structures in Python:

2.1 Conditions

2.1.1 The if statement for making decisions.

The if statement in Python is used to make decisions in your program based on specific conditions. It allows you to execute a block of code if a condition is evaluated as true, and possibly another block of code if the condition evaluates to false. Here is the general syntax of the if statement in Python:

```
if condition:
        # Block of code to execute if the condition is true (True)
else:
        # Block of code to execute if the condition is false (False)
```

The else part is optional, you can omit it if you don't want to execute any code when the condition is false.

Simple example to illustrate the use of the if statement:

```
age = 20
if age >= 18:
    print("You are an adult.")
else:
     print("You are a minor.")
You are an adult.
```

In this example, we use the if statement to check if age is greater than or equal to 18. If the condition is true, the program displays “You are an adult.” Otherwise, it displays “You are a minor.”

2.1.2 The elif instruction to manage several conditions.

There is also a variation of the if statement called elif (short for “else if”), which you allows you to specify multiple conditions to check in sequence. Here is an example :

```
[7]: score = 75
if note >= 90:
        print("A")
elif rating >= 80:
        print("B")
elif rating >= 70:
        print("C")
else:
        print("D")
C
```

In this example we check the grade and display a grade letter based on the range in which it is located. If the grade is greater than or equal to 90, “A” is displayed. Otherwise, if the grade is greater than or equal to 80, “B” is displayed, and so on. If none of the conditions are true, “D” is displayed.

The if statement is essential for controlling the execution flow of your program based on conditions and data provided. It allows you to create more dynamic programs and responsive depending on the situations encountered.

2.1.3 Practical condition exercises.

Exercise 1: Checking parity

Write a program that asks the user to enter an integer. Then use a if statement to determine whether the number is even or odd, and display an appropriate message.

Exercise 2: Calculation of the reduced price

Write a program that asks the user to enter the price of an item. If the price is greater than or equal to 10,000 F, apply a 10% reduction and display the new reduced price. Otherwise, display the price as is.

Exercise 3: Validating a password

Ask the user to enter a password. Then compare the entered password with a predefined password (for example, “password123”). If the two passwords match, display “Access Allowed”. Otherwise, display “Access Denied”.

Exercise 4: Evaluating a note

Ask the user to enter a rating between 0 and 100. Use an if statement to display a grade letter based on the grade according to the following criteria:

• If the grade is greater than or equal to 90, display “A”.

• If the grade is between 80 and 89, display “B”.

• If the grade is between 70 and 79, display “C”.

• If the grade is between 60 and 69, display “D”.

• If the grade is below 60, display “F”.

Exercise 5: Comparing numbers

Ask the user to enter two numbers. Then, compare the two numbers to determine if they are equal, if one is greater than the other, or if they are both equal. Show an appropriate message depending on the result.

2.2 Loops

Loops are control structures in programming that allow you to execute a block of code repeatedly as long as a specified condition is true (while loop) or for a predetermined number of iterations (for loop). Loops are a fundamental concept of programming and are widely used to automate repetitive tasks. In Python, You have two main types of loops: the while loop and the for loop.

2.2.1 The while loop to repeat instructions as long as a condition is true.

The while loop allows you to execute a block of code as long as a condition is true. Here is the syntax general:

while condition:

    # Block of code to execute as long as the condition is true

Example :

In this example, the while loop displays counter values as long as counter is lower at 5.

```
[2]: counter = 0
     while counter < 5:
         print(counter)
         counter += 1
```

```
0
1
2
3
4
```

2.2.2 The for loop to traverse elements of a sequence.

The for loop allows you to iterate over a sequence (such as a list, a tuple, or a string of characters) or generate a sequence of numbers from a specified range. Here is the syntax general:

for variable **in** sequence:

Block of code to execute for each element of the sequence

Example 1: Iteration over a list.

```
[3]: counter = 0
     while counter < 5:
         print(counter)
         counter += 1
```

```
0
1
2
3
4
```

Example 2: Generating a sequence of numbers.

```
[5]: for i in range(1, 6):
         print(i)
```

```
1
2
3
4
5
```

In these examples, the for loop iterates over the list of numbers and over the sequence generated by range(1, 6).

Break and continue instructions:

• The **break** statement is used to exit a loop prematurely if a certain condition is filled.

Example :

```
[12]: for i in range(0, 10):
          if i==5:
              break
          print(i)
```

```
0
1
2
3
4
```

This loop prints the numbers 1 to 4, then uses break to exit when i equals 5.

• The continue statement is used to move to the next iteration of a loop if a certain condition is filled, skipping the rest of the code in this iteration.

Example :

```
[20]: for i in range(1, 6):
          if i ==3:
             continue
          print(i)
```

```
1
2
4
5
```

This loop displays all numbers from 1 to 5 except 3.

Loops are a powerful tool for automating repetitive tasks and iterating over sequences data in Python. You can combine them with if statements to create structures complex controls in your programs.

2.2.3 Using loops to solve a problem

Loops are commonly used to solve a variety of problems in programming, especially when you need to repeat an action multiple times or iterate over data. Here are some examples of problems you can solve using loops in Python:

1. Calculation of the sum of numbers from 1 to n:

Suppose you want to calculate the sum of all numbers from **1** to **n** (where **n** is a number given). You can use a loop for to do this:

```
[1]: n = 10
     sum = 0
     for i in range(1, n + 1):
         sum += i
         print("The sum of numbers from 1 to", n, "is", sum)
```

```
The sum of numbers from 1 to 10 is 1
The sum of numbers from 1 to 10 is 3
The sum of numbers from 1 to 10 is 6
The sum of numbers from 1 to 10 is 10
The sum of numbers from 1 to 10 is 15
The sum of numbers from 1 to 10 is 21
The sum of numbers from 1 to 10 is 28
The sum of numbers from 1 to 10 is 36
The sum of numbers from 1 to 10 is 45
The sum of numbers from 1 to 10 is 55
```

2. Searching for an element in a list: If you have a list and want to check if a specific item is there, you can use a loop for to iterate through the list and perform the search:

```
[15]: list = [3, 6, 9, 12, 15]
      search_element = 9
      for element in list:
          if element == search_element:
              print("The element", search_element, "was found in the list.")
              break
          else:
              print("The element", search_element, "was not found in the list.")
```

```
The element 9 was not found in the list.
The element 9 was not found in the list.
The element 9 was found in the list.
```

3. Sequence generation: You can use loops to generate sequences of numbers or characters. By example, to generate the first ten perfect squares:

```
[16]: for i in range(1, 11):
          square = i ** 2
          print(f"The perfect square of {i} is {square}")
```

```
The perfect square of 1 is 1
The perfect square of 2 is 4
The perfect square of 3 is 9
The perfect square of 4 is 16
The perfect square of 5 is 25
The perfect square of 6 is 36
The perfect square of 7 is 49
The perfect square of 8 is 64
The perfect square of 9 is 81
The perfect square of 10 is 100
```

4. Factorial calculation:

Calculating the factorial of a number (denoted n!) is a classic example of using loops. The factorial of n is the product of all integers from 1 to n.

```
[18]: n = 5
      factorial = 1
      for i in range(1, n + 1):
          factorial *= i
      print(f"{n}! = {factorial}")
```

```
5! = 120
```

5. Iteration over elements of a character string:

You can use a for loop to iterate over characters in a string and perform various operations, such as counting the number of occurrences of a specific character:

```
[20]: sentence = "Hello world"
      counter = 0
      letter = "o"
      for char in sentence:
          if char == letter:
              counter += 1
              print(f"The character '{letter}' appears {counter} times in the sentence.")
```

```
The character 'o' appears 1 times in the sentence.
The character 'o' appears 2 times in the sentence.
```

Loops allow you to automate repetitive tasks and manipulate data from iterative manner, which is essential for solving it problems.

2.2.3 Practical loop exercises.

Exercise 1: Calculating the sum of integers

Write a program that calculates the sum of integers from 1 to 100 (inclusive) using a loop for. Then display the sum.

Exercise 2: Finding a multiple

Write a program that finds and displays all multiples of 7 between 100 and 200 (included) using a for loop.

Exercise 3: Multiplication table

Ask the user to enter an integer. Then write a program that displays the multiplication table of this number from 1 to 10 using a for loop.

Exercise 4: Calculating the average

Ask the user to enter a series of numbers (use a sentinel value to indicate the end of the entry, for example, -1). Then write a program that calculates the average of these numbers using a while loop.

Exercise 5: Palindrome

Ask the user to enter a string. Write a program that checks if the string is a palindrome (i.e. it reads the same from left to right and right to left, ignoring spaces and case) using a for loop.

Exercise 6: Calculation of the factorial

Ask the user to enter an integer. Write a program that calculates the factorial of this number using a for loop.

Exercise 7: Password management

Write a program that asks the user to enter a password. If the password entered incorrectly (for example, “password123”), ask the user to try again until he enters the correct password.

Exercise 8: Displaying Patterns

Write a program that displays the following patterns using nested for loops:

```
*
**
***
****
*****
1
12
123
1234
12345
A
BB
CCC
DDDD
EEEEE
```

2.3 Lists and Dictionaries

In Python, lists and dictionaries are two commonly used data structures for store and organize information. Here's how you can use them:

2.3.1 Lists

A list is an ordered collection of items, and each item can be of any data type. Lists are defined using square brackets [].

Example of creating a list and accessing its elements:

```
[21]: my_list = [1, 2, 3, 4, 5]
      print(my_list[0]) # Access the first element (index 0)

      1
```

```
[22]: my_list = [1, 2, 3, 4, 5]
      print(my_list[2]) # Access the third element (index 2)

      3
```

You can also add, remove, or modify items from a list:

my_list.append(6) # Add an element to the end of the list

```
[29]: my_list = [1, 2, 3, 4, 5]
      my_list.append(6) # Add an element to the end of the list
      print(my_list)

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

my_list.insert(2, 7) # Insert 7 at index 2

```
[30]: my_list = [1, 2, 3, 4, 5]
      my_list.insert(2, 7) # Insert 7 at index 2
      print(my_list)

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

my_list.remove(3) # Remove the first occurrence of 3

```
[28]: my_list = [1, 2, 3, 4, 5]
      my_list.remove(3) # Remove the first occurrence of 3
      print(my_list)

      [1, 2, 4, 5]
```

my_list.pop(4) # Delete the element at index 4

```
[31]: my_list = [1, 2, 3, 4, 5]
      my_list.pop(4) # Delete the element at index 4
      print(my_list)

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

my_list[1] = 10 # Modify an element

```
[32]: my_list = [1, 2, 3, 4, 5]
      my_list[1] = 10 # Modify an element
      print(my_list)

      [1, 10, 3, 4, 5]
```

List Creations In Python, you can create lists in different ways. Here is some common methods for creating lists:

Manual creation: You can create a list by specifying its elements in square brackets [].

For example :

my_list = [1, 2, 3, 4, 5]

Using the list() function: You can create a list from another sequence (like a string, a tuple, or even another list) using the list() function. For example :

string = "Python"

list_of_characters = list(string)

List Comprehensions: List comprehensions are a concise way to create lists in Python by applying an expression to each element in a sequence. For example, for create a list of squares of numbers 1 to 5:

squares = [x**2 for x in range(1, 6)]

```
[33]: squares = [x**2 for x in range(1, 6)]
      print(squares)

      [1, 4, 9, 16, 25]
```

Using the range() function: You can create a list of consecutive integers by using the range() function and converting the result to a list. For example, to create a list numbers from 0 to 9:

numbers = list(range(10))

Empty lists: You can create an empty list using [] or using the constructor list(). For example :

empty_list = []

other_empty_list = list()

Concatenating lists: You can create a list by concatenating two or more lists existing. For example:

list1 = [1, 2, 3]

list2 = [4, 5, 6]

new_list = list1 + list2

Repeating lists: You can create a list by repeating an existing list several times using the * operator. For example :

original_list = [1, 2]

repeated_list = original_list * 3 # Creates a list [1, 2, 1, 2, 1, 2]

Accessing List Elements To access elements of a list in Python, you use the index of the element. Subscripts start at 0 for the first element and increase by 1 for each following item. Here's how to add elements to a list:

Accessing an element by index: You can access a specific element using its index in square brackets []. For example, for the first element of the list my_list:

[16]: my_list = [10, 20, 30, 40, 50]

first_element = my_list[0] # Access to the first element (index 0)

print(first_element) # This will print 10

10

Access to an element by negative index: You can also use negative indices to access items from the end of the list. For example, -1 refers to the last element, -2 to the second element from the end, and so on:

[17]: last_element = my_list[-1] # Access to the last element

penultimate_element = my_list[-2] # Access to penultimate element

Slicing: You can extract a portion (or slice) from the list by specifying a index range. Slicing is performed using: between the start and end indices (the last specified index is not included). For example :

[18]: my_list = [10, 20, 30, 40, 50]

sub_list = my_list[1:4] # Creates a new list with elements 20, 30 and 40

Accessing multiple items: You can also access multiple items using indices or ranges of indices in a list.

For example :

[20]: my_list = [10, 20, 30, 40, 50]

elements = [my_list[0], my_list[2], my_list[4]] # Access to elements␣specific [10, 30, 50]

Accessing the elements of a list is fundamental in programming because it allows you to manipulate the data stored in the list efficiently. Make sure to use valid indices to avoid out-of-range index errors.

Editing Lists In Python, you can edit lists in several ways, including adding, removing, or modifying elements. Here are some common operations for modifying lists:

Add items:

• append(): Adds an element to the end of the list.

my_list = [1, 2, 3]

my_list.append(4) # Adds 4 to the end of the list

• insert(): Inserts an element at a specific location using an index.

my_list = [1, 2, 3]

my_list.insert(1, 4) # Inserts 4 at index 1 (between 1 and 2)

• List extension (+= or extend()): You can add the elements of another list to the end of existing list.

my_list = [1, 2, 3]

other_list = [4, 5]

my_list += other_list # Equivalent to my_list.extend(other_list)

Delete items:

• remove(): Removes the first occurrence of an element by its value.

my_list = [1, 2, 3, 2]

my_list.remove(2) # Removes the first occurrence of 2

• pop(): Removes an element at a specific index and returns it.

my_list = [1, 2, 3]

element_delete = my_list.pop(1) # Delete the element at index 1 (2) and return it

• del: Deletes an element or a range of elements by index.

my_list = [1, 2, 3, 4, 5]

del my_list[2] # Delete the element at index 2 (3)

Edit items:

• You can simply reassign a new value to an element using its index.

my_list = [1, 2, 3]

my_list[1] = 4 # Replaces the element at index 1 (2) with 4

• You can also use slicing to modify a range of elements.

my_list = [1, 2, 3, 4, 5]

my_list[1:3] = [6, 7] # Replaces elements 2 and 3 with 6 and 7

Useful list methods

Python provides a set of predefined methods for lists that are very useful for performing common list operations. Here are some of the most useful list methods:

• append(): Adds an element to the end of the list.

my_list = [1, 2, 3]

my_list.append(4) # my_list becomes [1, 2, 3, 4]

• extend(): Extends the list by adding the elements of another list at the end.

my_list = [1, 2, 3]

other_list = [4, 5]

my_list.extend(other_list) # my_list becomes [1, 2, 3, 4, 5]

• insert(): Inserts an element at a specific index.

my_list = [1, 2, 3]

my_list.insert(1, 4) # my_list becomes [1, 4, 2, 3]

• remove(): Removes the first occurrence of an element by its value.

my_list = [1, 2, 3, 2]

my_list.remove(2) # my_list becomes [1, 3, 2]

• pop(): Removes an element at a specific index and returns it.

my_list = [1, 2, 3]

element_supprime = my_list.pop(1) # my_list becomes [1, 3] and element_supprime is 2

• index(): Returns the index of the first occurrence of an element by its value.

my_list = [10, 20, 30, 20]

index = my_list.index(20) # index becomes 1

• count(): Returns the number of occurrences of an element in the list.

my_list = [1, 2, 2, 3, 2]

occurrences = my_list.count(2) # occurrences becomes 3

• sort(): Sorts the existing list in ascending (default) or descending order.

my_list = [3, 1, 2]

my_list.sort() # my_list becomes [1, 2, 3]

• reverse(): Reverses the order of the list elements.

my_list = [1, 2, 3]

my_list.reverse() # my_list becomes [3, 2, 1]

• copy(): Creates a shallow copy of the list.

my_list = [1, 2, 3]

copy = my_list.copy() # Creates an independent copy of my_list

These methods make it easier to manipulate and modify lists in Python. They are very useful in many programming scenarios.

2.3.2 Dictionaries

Introduction to Dictionaries and Their Key-Value Structure

A Dictionary in Python is a data structure that allows information to be stored and organized in a manner flexible using a key-value relationship. Unlike lists, which are indexed by numerical indices, dictionaries are indexed by keys. Every element in a dictionary consists of a key-value pair, where the key is unique and provides access to the associated value.

Creating a dictionary:

Dictionaries are defined using braces {} or the dict() function. The key-pairs value are separated by colons:, and pairs are separated by commas.

my_dictionary = {"key1": "value1", "key2": "value2", "key3": "value3"}

Access to values:

To access a value in a dictionary, you use the corresponding key in square brackets []:

value = my_dictionary["key1"] # Accesses the value associated with "key1"

Changing values:

You can change the value associated with a key by reassigning a new value to that key: my_dictionary["key2"] = "new_value" # Modifies the value associated with "key2"

Adding new key-value pairs:

You can add new key-value pairs to a dictionary by specifying a new key and its value:

my_dictionary["new_key"] = "new_value" # Adds a new key-value pair

Deleting key-value pairs:

You can delete a key-value pair using the del statement:

del my_dictionary["key3"] # Delete the key-value pair associated with "key3"

Useful Dictionary Methods:

• keys(): Returns a list of dictionary keys.

• values(): Returns a list of dictionary values.

• items(): Returns a list of key-value pairs as tuples.

• get(): Returns the value associated with a key, handling non-existent keys.

• in (membership operator): Used to check if a key exists in the dictionary.

Example of using a dictionary in Python:

```
[21]: personal_information = {
        "name": "Bio",
        "first name": "Mourou",
        "age": 26,
        “city”: “Mbour”
}
print(personal_information["name"]) # Displays "Bio"

print(personal_information["age"]) # Poster 26

personal_information["profession"] = "Engineer" # Add a new ⌴↳key-value pair

del personal_information["city"] # Delete the "city" key
```

Dictionaries are extremely useful for storing and retrieving data associated with keys, which makes them suitable for many applications, such as structured data management, configuration settings, database query results, etc.

2.3.3 Practical exercises with lists and dictionaries.

Exercise 1: Lists

1. Create a list called numbers containing the integers 1 to 10.

2. Add the number 11 to the end of the list.

3. Use a for loop to iterate through the list and display each number.

4. Remove odd numbers from the list.

5. Display the sum of the remaining numbers in the list.

Exercise 2: Dictionaries

1. Create a dictionary called personal_information containing the following information about a person: first name, last name, age, and city.

2. Add the key “profession” to the dictionary and associate it with the value “Engineer”.

3. Display the person's name.

4. Change the person's age in the dictionary.

5. Delete the “city” key from the dictionary.

Exercise 3: Data fusion

1. Create two lists: names and scores. Lists contain player names and their scores respective.

2. Create a dictionary called ranking by combining the two lists in such a way that the names are the keys and the scores are the associated values.

3. View the leaderboard sorted in descending order of scores.

Exercise 4: Product management

1. Create a dictionary called product representing a product with the following keys: “name”, “price”, “quantity”.

2. Display the product name.

3. Calculate the total cost of the product (price * quantity) and display it.

4. Change the quantity of the product in the dictionary.

5. Create a list of multiple products (at least three) and store them in a list called product_list, then display it.

6. Use a loop to calculate the total cost of all products in the list.

3 Scientific and Statistical Libraries

3.1 Introduction to libraries

In Python, a library (or module) is a set of functions, classes and variables predefined which allow you to extend the functionalities of the base language. These libraries are grouped into packages, which are directories containing a set of modules linked by a common functionality. Libraries are essential in Python because they allow developers to access a wide range of features without having to reinvent the wheel.

Some important characteristics of libraries in Python:

1. Code Reuse: Libraries provide pre-written functions and classes that developers can use in their programs. This allows code to be reused existing and accelerate the development process.

2. Extended Features: Libraries extend the capabilities of Python in various areas, such as data processing, file management, interface creation user, network communication, data science, machine learning, etc.

3. Modularity: The libraries are designed in a modular way, which means that you can choose the appropriate libraries for your project and import them only when you need it. This helps optimize the use of resources.

4. Community and support: Python has an active development community that creates and maintains numerous libraries. This means you can usually find a library to solve almost any problem you might encounter.

5. Ease of use: The libraries are designed to be user-friendly, making it easier their use by developers. They are generally well documented with examples of use. To use a library in Python, you must import it into your code using the import statement. For example, to import the math library (which contains functions mathematics), you can write:

import math

You can now use functions from the math library, for example: result = math.sqrt(25) # Calculate the square root of 25

Python has a rich standard library that ships with the language and contains many libraries for common tasks. Additionally, there are a multitude of third-party libraries developed by the Python community for more specific needs. You can install these third-party libraries using tools like pip to use them in your projects.

3.2 Introduction to NumPy

NumPy, short for “Numerical Python”, is a fundamental Python library for numerical and scientific calculation. It provides data structures and functions for working with multidimensional arrays and matrices, as well as tools for performing advanced mathematical and statistical operations. NumPy is widely used in the field data science, machine learning, mathematical modeling, engineering, and many other fields where digital calculation is necessary.

3.2.1 Presentation of NumPy as a library for manipulating multidimensional arrays.

NumPy (Numerical Python) is a fundamental library in Python that stands out for its ability to efficiently manipulate multidimensional arrays, called “ndarrays”. NumPy offers a powerful feature set for creating, manipulating, processing and analyzing data in the form of multidimensional tables.

Creating NumPy Arrays:

Creating NumPy arrays is simple and flexible. You can create arrays from Python lists, tuples, sequences, files, or by using special NumPy creation functions : import numpy as np

Creating a table from a list

arr1 = np.array([1, 2, 3, 4, 5])

Creating an array of zeros

arr2 = np.zeros((3, 4)) # Creates a 3x4 array filled with zeros

Creating an array of random numbers

arr3 = np.random.rand(2, 2) # Creates a 2x2 array of random numbers between 0 and 2

Creating multidimensional tables

You can create multidimensional tables by providing nested lists or tuples of appropriate dimensions during creation.

2D_array = np.array([[1, 2, 3], [4, 5, 6]]) # Creates a 2D array

3D_array = np.array([[[1, 2], [3, 4]], [[5, 6], [7, 8]]])

Array Manipulation: NumPy offers many functions for performing operations on arrays, including slicing, resizing, sorting, concatenation and many others.

Access to elements in an array

element = arr1[2] # Access to element at index 2

Resizing a table

arr4 = arr1.reshape((1, 5)) # Resize the array to 1x5

Sorting an array

arr5 = np.sort(arr1) # Sort the array in ascending order

Vector and mathematical operations

NumPy allows you to perform vector and mathematical operations on arrays, which greatly simplifies numerical calculations.

Vector operations result = arr1 * 2 # Multiply each element by 2

Mathematical operations

mean = np.mean(arr1) # Calculate the average of the elements

sum = np.sum(arr1) # Calculate the sum of the elements

Advanced indexing

NumPy supports advanced indexing of arrays, which allows you to select items based on specific criteria.

Select items greater than 3

elements_sup_3 = arr1[arr1 > 3]

Statistics and linear algebra functions:

NumPy includes a wide range of functions for statistics, linear algebra, Fourier transforms, etc.

Calculation of mean and covariance

mean = np.mean(arr1)

covariance = np.cov(arr1, arr2)

#Matrix multiplication

matrix_result = np.dot(matrix1, matrix2)

NumPy is an essential library for manipulating multidimensional arrays in Python. It offers high performance, user-friendly syntax and a wide range of features for processing numerical and scientific data. This is why NumPy is widely used in fields such as data science, machine learning, numerical simulation, mathematical modeling and many others.

3.2.2 Practical examples with Numpy

Of course, here are two practical examples of using NumPy:

Example 1:

Calculating statistics with NumPy. Suppose you have a list of students' grades and you want to calculate the average, the median and standard deviation of these scores. Here's how you could do it with NumPy:

[22]: import numpy as np # Student Notes List marks = [85, 90, 78, 92, 88, 76, 85, 89, 94, 90]

Converting the list to a NumPy array

notes_array = np.array(notes)

Calculation of the average

mean = np.mean(notes_array)

Calculation of the median

median = np.median(notes_array)

Calculation of standard deviation

standard_dev = np.std(notes_array)

print("Average:", average)

print("Median:", median)

print("Standard deviation:", standard_deviation)

Average: 86.7

Median: 88.5

Standard deviation: 5.532630477449222

Example 2:

Vector operations with NumPy Suppose you have two lists of data, for example, the sales of two products during of four months, and you want to calculate the total income for each month by performing a vector operation. Here's how you could do it with NumPy:

[23]: import numpy as np

Sales data for two products (in thousands of dollars) for each month

product_A = [50, 60, 75, 80]

product_B = [40, 55, 70, 85]

Converting lists to NumPy arrays

sales_A = np.array(product_A)

sales_B = np.array(product_B)

Calculation of total income for each month by adding the sales of both ↪products

total_revenue = sales_A + sales_B

print("Total income for each month:", total_income)

Total income for each month: [90 115 145 165]

3.2.3 Practical exercises with NumPy

Exercise 1: Calculations with NumPy

1. Create a NumPy array containing the numbers 1 to 10.

2. Calculate the sum, mean, median and variance of this table.

3. Replace all even numbers in the table with their squares.

Exercise 2: Manipulation of matrices

1. Create two 2x2 NumPy matrices, matrix1 and matrix2, with values of your choice.

2. Perform matrix multiplication of the two matrices.

3. Calculate the transpose of matrix1.

4. Add the two matrices.

Exercise 3: Generating random data

1. Generate a NumPy array of 10 random numbers between 0 and 1.

2. Create a NumPy array of 100 random numbers following a normal distribution (law Gaussian) with a mean of 5 and a standard deviation of 2.

3. Calculate the mean, variance and standard deviation of the table generated in the second part of exercise.

Exercise 4: Advanced manipulation of tables

1. Create a 1D NumPy array with integers from 1 to 20.

2. Filter the table to get only numbers divisible by 3.

3. Use the numpy.linspace() function to create a NumPy array of 10 numbers equidistant between 0 and 1.

4. Create an array of the shape (4, 4) with increasing values from 1 to 16, filling in first each column before moving on to the next one.

3.3 Introduction to Matplotlib

3.3.1 Presentation of Matplotlib as a library for creating graphics and visualizations.

Matplotlib is a widely used Python data visualization library that allows create a wide variety of graphs, charts and visualizations to represent and interpret data. It offers exceptional flexibility to customize the design and the style of your graphics.

Main features of Matplotlib: Creating charts in 2D and 3D: Matplotlib supports charting two-dimensional (2D) and three-dimensional (3D) graphics, which makes it possible to visualize data in various spaces.

Supported chart types: Matplotlib offers a diverse range of graphics, including bar charts, line charts, pie charts, scatter plots, histograms, box plots, dispersion, heat maps, and much more.

Graphics Customization: You can customize every aspect of your graphics, from colors and line styles to labels, captions, titles, axes and annotations. Matplotlib offers complete control over the appearance of your visualizations.

Export and integration: You can export your charts in various formats file, including PNG, PDF, SVG and others, for inclusion in reports, presentations or publications. Matplotlib also allows you to integrate graphs into Python graphical user interfaces (GUIs).

Interactivity: Matplotlib can be used in conjunction with interactivity libraries like mpld3 or Bokeh to create interactive graphics in web browsers.

Basic use of Matplotlib:

Here is a simple example of creating a line graph with Matplotlib:

```
import matplotlib.pyplot as plt

# Data

x = [1, 2, 3, 4, 5]

y = [10, 15, 13, 18, 25]

# Creation of the graph

plt.plot(x, y)

# Adding labels and a title

plt.xlabel("X-Axis")

plt.ylabel("Y-Axis")

plt.title("Simple Linear Graph")

# Graph display

plt.show()
```

In this example, we used Matplotlib's pyplot module to create a graph linear. We specified the x and y data, added labels and a title, then displayed the graph on the screen with plt.show().

Matplotlib is a versatile and powerful library that finds applications in many fields, including data science, scientific research, data analysis, data visualization, generating post charts, mapping, and more Moreover. It has become a reference in data visualization in Python.

3.3.2 Creating simple graphs with matplotlib

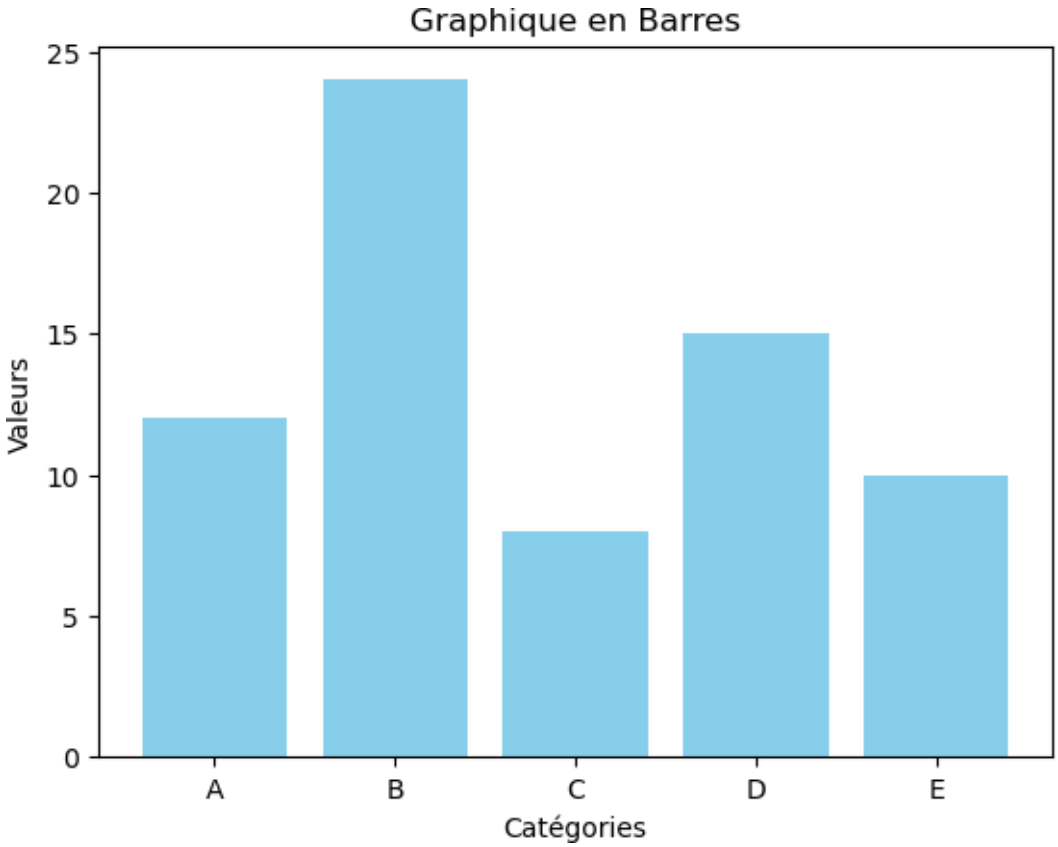
Example 1: Bar Chart

```
[24]: import matplotlib.pyplot as plt

# Data

categories = ['A', 'B', 'C', 'D', 'E']

values = [12, 24, 8, 15, 10]

# Creating the bar chart

plt.bar(categories, values, color='skyblue')

# Adding labels and a title

plt.xlabel("Categories")

plt.ylabel("Values")

plt.title("Bar Chart")
```

Graph display

plt.show()

Matplotlib is building the font cache; this may take a moment.



Example 2: Linear Graph

[25]: import matplotlib.pyplot as plt

Data

x = [1, 2, 3, 4, 5]

y = [10, 15, 13, 18, 25]

Creation of the line graph

plt.plot(x, y, marker='o', linestyle='-', color='green')

Adding labels and a title

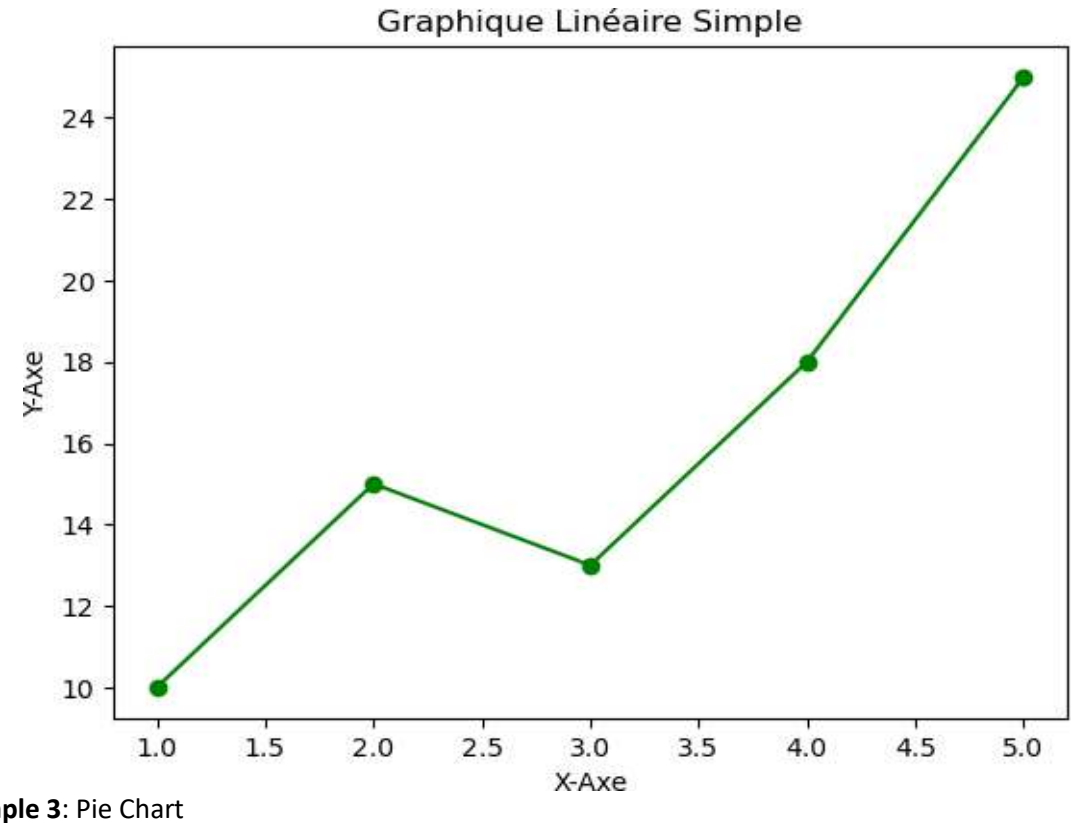
plt.xlabel("X-Axis")

plt.ylabel("Y-Axis")

plt.title("Simple Linear Graph")

Graph display

plt.show()



Example 3: Pie Chart

[26]: import matplotlib.pyplot as plt

Data

labels = ['A', 'B', 'C', 'D']

percentages = [30, 20, 25, 25]

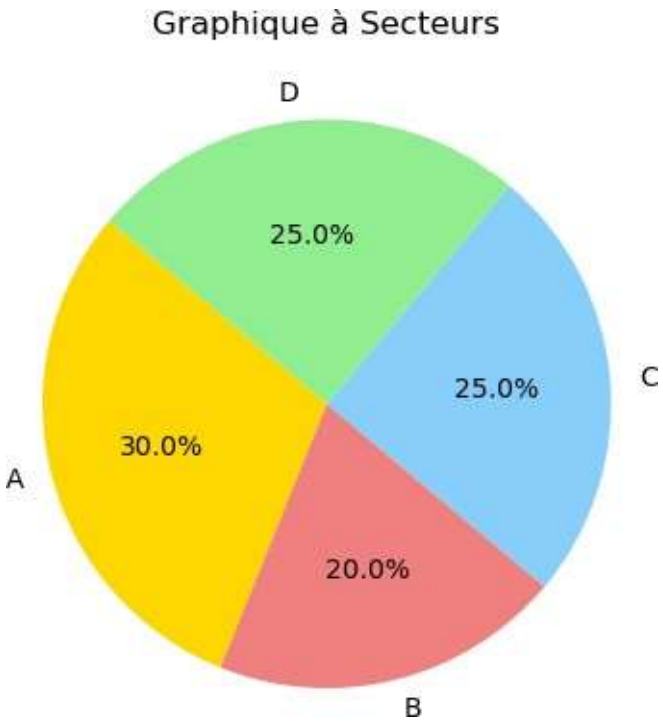
colors = ['gold', 'lightcoral', 'lightskyblue', 'lightgreen']

Creation of the pie chart

plt.pie(percentages, labels=labels, colors=colors, autopct='%1.1f%%', startangle=140)

Adding a title plt.title("Pie Chart")

Graph display plt.show()



These examples show how to create bar charts, line charts, and Basic pie charts using Matplotlib. You can further customize these graphics by adjusting colors, line styles, labels, legends and more settings to meet your specific needs. Matplotlib offers great flexibility for creating custom graphics.

3.3.3 Personalization of graphics

Chart customization is one of the key features of Matplotlib. She you allows you to control the appearance and style of your graphics to suit your specific needs. cifics. Here are some of the many customization options you can use with Matplotlib to create custom charts:

Colors: You can specify the color of your chart elements using the color argument in plot functions. For example, color='red' will set the color of the drawn in red.

Line Styles: You can specify the line style with the linestyle argument. By example, linestyle='--' creates a dotted line.

Markers: For line charts, you can add markers to your points of view. data using the marker argument. For example, marker='o' will add circles to points.

Line Width: You can set the line width with the linewidth argument. By example, linewidth=2 will make the line twice as thick.

Axis Labels: You can add labels to the X and Y axes using the functions xlabel and ylabel. For example, plt.xlabel("X-Axis").

Title: You can add a title to the chart using the title function. For example, plt.title("My Chart").

Legend: If you have several data series in the same chart, you can add a legend using the legend function. You must also specify labels for each series.

Axis Limits: You can define the limits of the X and Y axes with the xlim and ylim.

Grid: You can add a background grid to your chart using the grid function.

Data Labels: You can add labels to data points using the text function. This is useful for annotating specific points.

Color Palette: You can use color palettes for plots of several data series using the set_prop_cycle function.

Figure Size: You can set the size of the figure using the figure function. For example, plt.figure(figsize=(8, 6)) will define a figure that is 8 inches wide and 6 inches height. Example of customizing a bar chart: This is just one example of customization among many other options available with Matplotlib. You can further explore Matplotlib documentation to customize your graphics based on your specific needs.

[28]: import matplotlib.pyplot as plt

Data

categories = ['A', 'B', 'C', 'D']

values = [12, 24, 8, 15]

Creating the custom bar chart

plt.bar(categories, values, color='skyblue', edgecolor='black', linewidth=2)

Adding labels and a title

plt.xlabel("Categories")

plt.ylabel("Values")

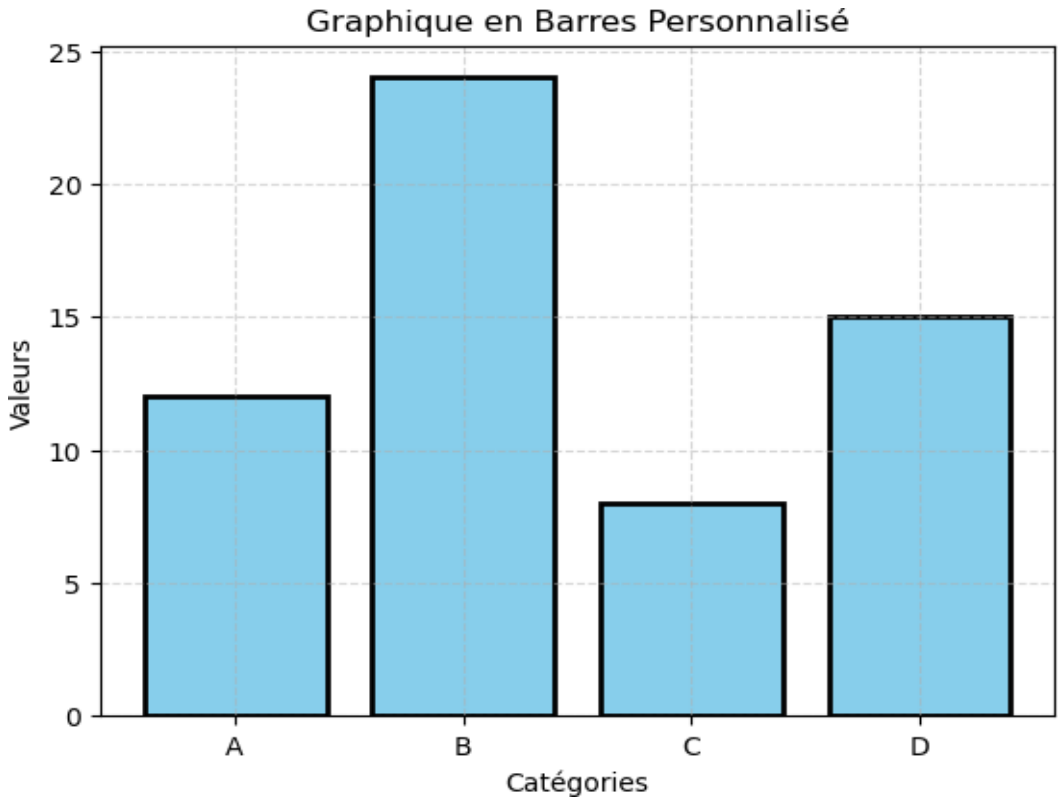
plt.title("Custom Bar Chart")

Added a grid

plt.grid(True, linestyle='--', alpha=0.5)

Graph display

plt.show()



3.3.4 Practical examples with Matplotlib.

Example 1: Stacked Bar Chart Suppose you have sales data for two products (A and B) over the course of four quarters, and want to create a stacked bar chart to visualize sales totals of each product per quarter.

[29]: import matplotlib.pyplot as plt

Data

quarters = ['Q1', 'Q2', 'Q3', 'Q4']

sales_product_A = [50, 60, 75, 80]

sales_product_B = [40, 55, 70, 85]

Creating the stacked bar chart

plt.bar(quarters, sales_product_A, label='Product A', color='blue')

plt.bar(quarters, sales_product_B, label='Product B')

  bottom=sales_product_A, color='orange')

Adding labels and a title

plt.xlabel("Quarters")

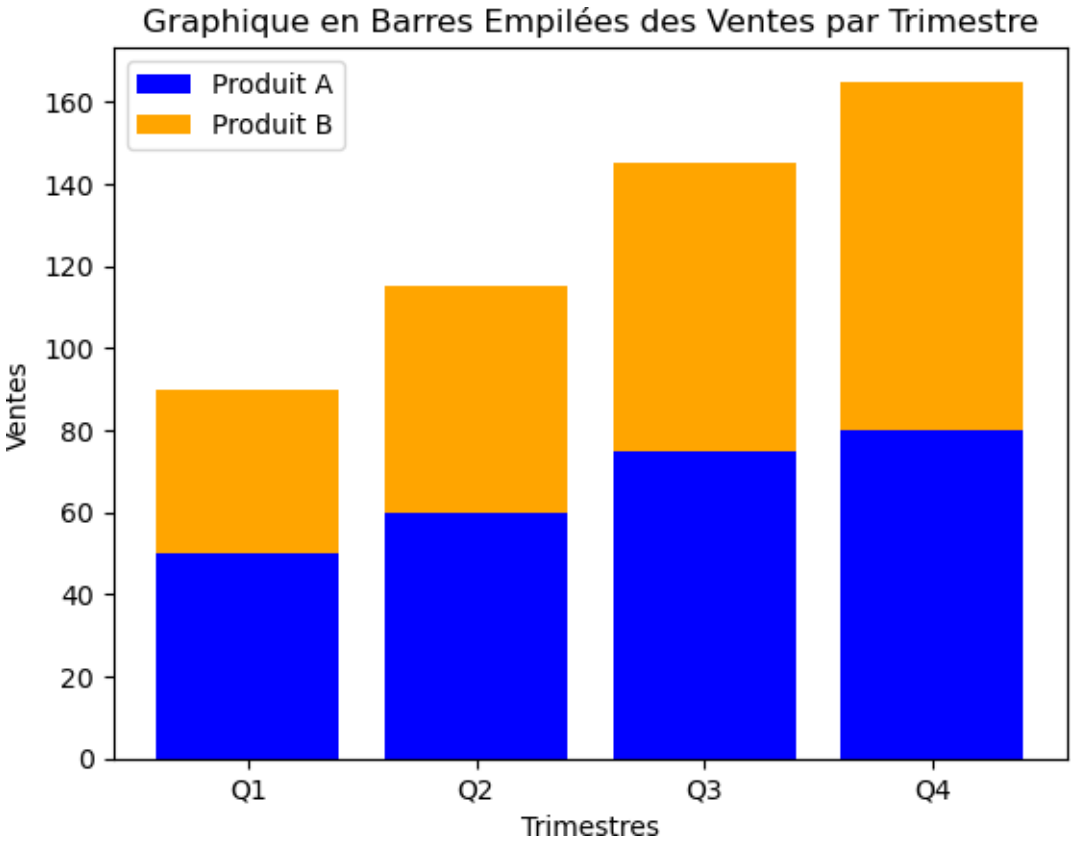
plt.ylabel("Sales")

plt.title("Stacked Bar Chart of Sales by Quarter")

plt.legend()

Graph display

plt.show()



Example 2: Scatter Plot

Suppose you have height and weight data from several people, and you wanted to create a scatterplot to visualize the relationship between these two variables.

[30]: import matplotlib.pyplot as plt

Data

size = [160, 165, 170, 175, 180, 185]

weight = [60, 65, 70, 75, 80, 85]

Creation of the scatter diagram plt.scatter(size, weight, color='green', marker='o', label='Data')

Adding labels and a title

plt.xlabel("Size (cm)")

plt.ylabel("Weight (kg)")

plt.title("Size vs. Weight Scatter Plot")

plt.legend()

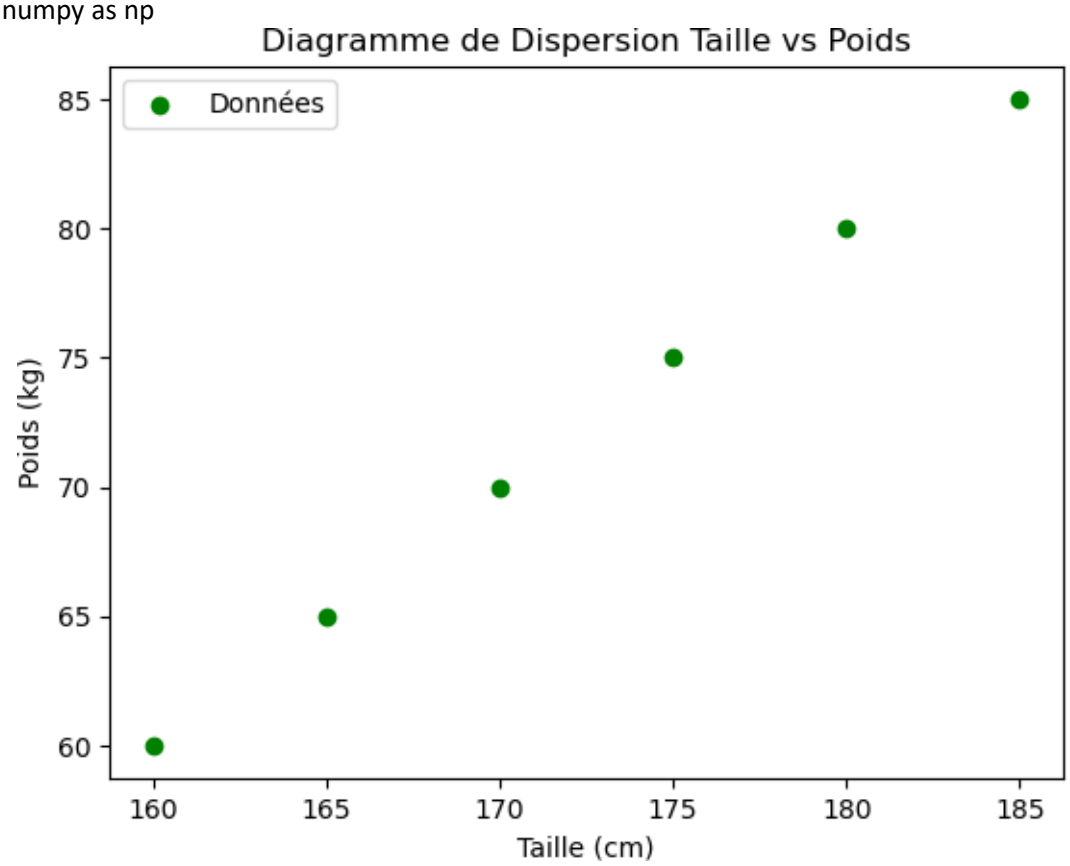
Graph display

Example 3: Histogram

Let's say you have data on a website's response times, and you want to create a histogram to visualize the distribution of response times.

[31]: import matplotlib.pyplot as plt

import numpy as np



Data (response time in milliseconds)

response_time = np.random.normal(300, 50, 1000)

Creation of the histogram

plt.hist(response_time, bins=20, color='purple', edgecolor='black')

Adding labels and a title

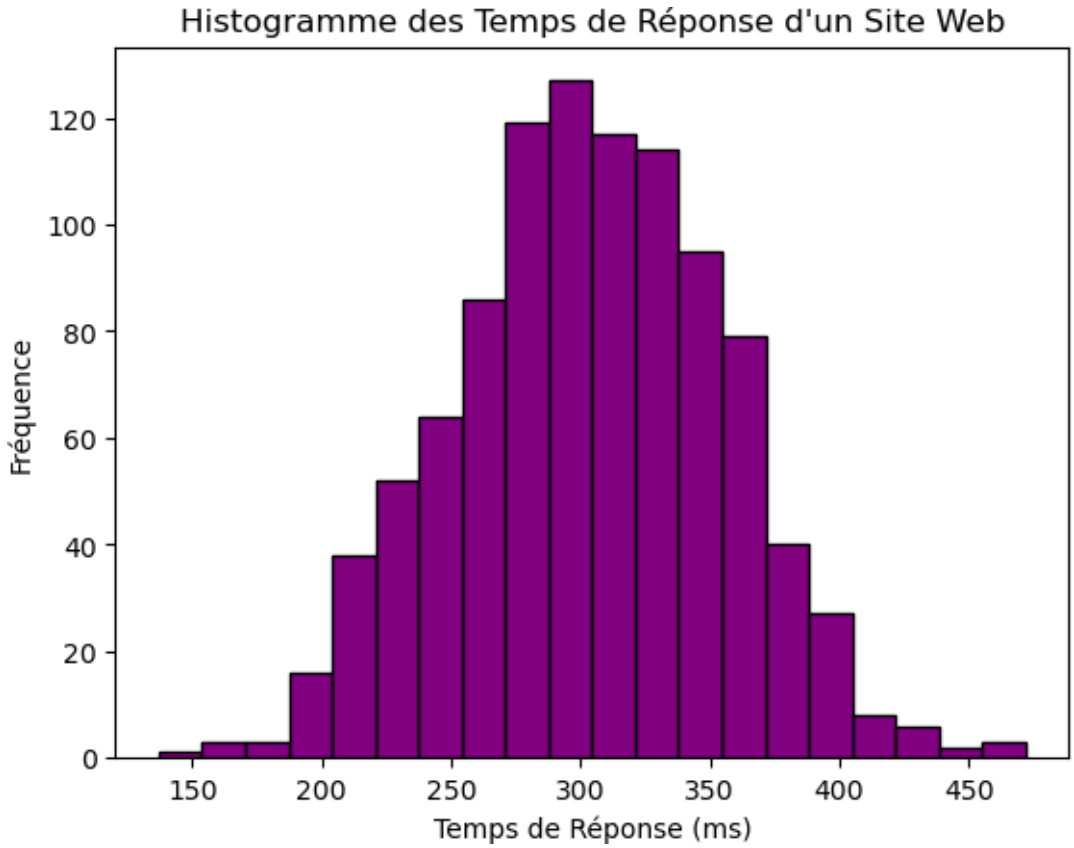
plt.xlabel("Response Time (ms)")

plt.ylabel("Frequency")

plt.title("Histogram of Website Response Times")

Graph display

plt.show()



3.3.5 Practical exercises with Matplotlib.

Exercise 1: Histogram of Notes

• Create a histogram from a dataset of student grades (for example, 100 notes between 0 and 100).

• Customize the histogram by choosing colors, adding axis labels and a title, and adjusting the number of bins to obtain an informative visualization.

Exercise 2: Market Share Pie Chart

• Suppose you have data on the market shares of different companies (e.g. example, Apple, Google, Microsoft, Amazon) in the form of percentages.

• Create a pie chart to visualize these market shares.

• Customize the chart by adding labels and highlighting the share of largest market.

Exercise 3: Stacked Bar Chart of Monthly Sales

• Use monthly sales data for two products (for example, Product A and Product B) over a full year.

• Create a stacked bar chart to display the total monthly sales of both products.

• Customize the chart by adding colors, axis labels, title and legend.

Exercise 4: Growth Data Scatter Plot

• Have data on the annual population growth of different cities during several years.

• Create a scatterplot to visualize the relationship between time (years) and population growth for each city.

• Personalize the diagram by choosing suitable colors and markers, adding axis labels and a title.

Exercise 5: Line Chart with Two Data Series

• Use data from two time series (for example, monthly sales of two products) over a period of time.

• Create a line chart to display both data series on the same chart.

• Customize the chart by adding different colors for each series, axis labels, a title and a legend.

3.4 Introduction to Pandas

3.4.1 Presentation of Pandas as a library for manipulation and data analysis.

Pandas is a popular and powerful Python library for manipulating and analyzing data. It offers flexible and efficient data structures for working with data tabular, such as tables or spreadsheets. Pandas is widely used in the field of data analysis, data science and data manipulation in various applications. Here is a presentation of Pandas as a library for data manipulation and analysis:

Pandas Key Features:

Data Structures: Pandas offers two main data structures: DataFrames and Series. - DataFrame: This is a two-dimensional tabular data structure similar to a spreadsheet or database table. DataFrames allow you to store data in a rectangular format where each column can have a different data type. - Series: This is a one-dimensional data structure similar to an array or list. Series are used to store data in a single column or a single dimension.

Reading and Writing Data: Pandas supports reading and writing data from various file formats, such as CSV, Excel, SQL, JSON, HDF5, and many more. This makes it easy to import and export data from and to different sources.

Indexing and Selection: Pandas offers powerful methods for indexing, selection and data filtering. You can access data using column labels, row indices, or boolean conditions.

Data Manipulation: Pandas allows you to perform various data manipulation operations. data, including grouping, sorting, aggregating, merging, pivoting and transforming Datas. This makes it easier to prepare data for analysis.

Management of Missing Values: Pandas offers tools to effectively manage values missing (NaN) data, including cleaning, padding, and interpolation.

Data Analysis: Pandas facilitates exploratory data analysis (EDA) by providing functions for calculating descriptive statistics, visualizing data, creating pivot tables and more.

3.4.2 Creation of Pandas data structures: DataFrames and Series.

The main data structures of the Pandas library are DataFrames and Series. These structures are used to manipulate and analyze tabular data efficiently. Here is an introduction to these two data structures:

Series: A Series is a one-dimensional data structure similar to an array or a column in a spreadsheet. It can contain data of different types (numbers, character strings, dates, etc.) and is associated with an index, which can be automatically generated or specified by the user. A Series looks like this:

```
import pandas as pd

data = pd.Series([10, 20, 30, 40, 50])
```

In this example, data is a Series with numeric values and a default index of 0 to 4.

Operations you can perform on a Series include element selection, math operations, filtering, and more. For example :

```
print(data[2]) # Access an element by its index (30)

print(data.mean()) # Calculate the mean (30.0)

print(data > 25) # Filter elements greater than 25
```

DataFrames: A DataFrame is a two-dimensional data structure similar to a database table or Excel spreadsheet. It is made up of rows and columns, where each column can be a Series. You can create a DataFrame from multiple data sources, such as dictionaries, CSV files, Excel, etc.

Here is an example of creating a DataFrame from a dictionary:

```
[43]: import pandas as pd

data = {
    'Name': ['Alice', 'Bob', 'Charlie'],
    'Age': [25, 30, 35],
    'City': ['Paris', 'New York', 'London']
}

df = pd.DataFrame(data)
```

The result is a DataFrame with three columns: ‘Name’, ‘Age’ and ‘City’, and three rows of data. You can perform many operations on DataFrames, such as selecting columns, filtering, sorting, grouping, arithmetic operations, etc.

[44]: print(df['Name']) # Select a column

print(df[df['Age'] > 28]) # Filter rows where age is greater than 28

0 Alice

1 Bob

2 Charlie

Name: Name, dtype: object

Name Age City

1 Bob 30 New York

2 Charlie 35 London

Series are one-dimensional data structures, while DataFrames are two-dimensional data structures. Pandas offers great flexibility to manipulate and analyze tabular data, making it one of the most popular libraries for data processing in Python.

3.4.3 Data manipulation with Pandas (filtering, sorting, grouping, etc.).

Pandas offers many features for manipulating tabular data, such as filtering, sorting, grouping and much more. Here are some common operations that you can do with Pandas:

Data Filtering: You can filter data based on certain conditions. By example, if you have a DataFrame df, you can filter rows where a specific column meets a given condition:

Filter rows where the 'Age' column is greater than 25

df_filtered = df[df['Age'] > 25]

Sorting data: You can sort data based on the values of one or more columns. Use the sort_values() method for this:

Sort the DataFrame in ascending order of age

df_sorted = df.sort_values(by='Age')

To sort by multiple columns, you can specify a list of columns:

Sort first by 'Age' then by 'Name' (in that order)

df_sorted = df.sort_values(by=['Age', 'Name'])

Grouping data: The groupby() function allows you to group data based on the values of one or more columns, then apply aggregate operations on each group. For example, to calculate the average age by city:

```
# Group by the 'City' column and calculate the average age in each group

average_age_by_city = df.groupby('City')['Age'].mean()
```

Data Aggregation: You can perform various aggregation operations such as sum, average, count, etc., on grouped data. For example, to obtain the number of people per city:

```
# Group by the 'City' column and count the number of rows in each group

number_of_people_per_city = df.groupby('City').size()
```

Reindexing: You can reorganize the index of your DataFrame with the reindex() method. This can be useful for rearranging rows or filling in missing values.

```
# Reorganize the DataFrame index

df = df.reindex([2, 0, 1])
```

Column manipulation: You can add, delete or rename columns in a DataFrame.

```
# Add a new column

df['New_Column'] = [True, False, True]

# Delete a column

del df['New_Column']

# Rename a column

df.rename(columns={'Age': 'Age'}, inplace=True)
```

These are just some of the basic operations you can do with Pandas. There library offers many other advanced features for cleaning, transforming and analyzing data, making it a powerful tool for data manipulation tabulars in Python.

3.4.4 Practical examples with Pandas.

Here is a simple example that illustrates the use of Pandas for manipulation and analysis of data. Suppose you have a set of product sales data, stored in a CSV file called “sales.csv”.

```
import pandas as pd

# Reading data from CSV

file sales_data = pd.read_csv("sales.csv")

# Displaying the first rows of the DataFrame

print("Data preview:")

print(sales_data.head())
```

```
# Calculation of sales average

average_sales = data_sales['Total Sales'].mean()

print("\nSales average:", sales_average)

# Filtering sales above 1000

sales_sup_1000 = sales_data[sales_data['Total Sales'] > 1000]

print("\nSales above 1000:\n", sales_sup_1000)

# Descriptive statistics of product prices

price_statistics = sales_data['Product Price'].describe()

print("\nProduct price statistics:\n", price_statistics)
```

In this example, we used Pandas to:

1. Read data from CSV file into a DataFrame.

2. Preview the first rows of the DataFrame with head().

3. Calculate the average of the total sales (Total Sales) using the mean() method.

4. Filter sales with a total amount greater than 1000 using a Boolean condition.

5. Display descriptive statistics of product prices using the describe() method.

You can further customize these operations based on your specific analysis needs. Pandas offers great flexibility for data manipulation, exploration and analysis tabular.

Pandas is an essential library for anyone working with tabular data in Python, whether for data analysis, data preparation, visualization or data modeling. It is often used in conjunction with other libraries like NumPy, Matplotlib and Scikit-Learn for more advanced data analysis tasks and machine learning.

3.4.5 Practical Exercises

Here are some exercises for applying Pandas concepts to the CSV file “etudiants.csv”.

Exercise 1: Score Statistics

• Calculate the mean, median, standard deviation and variance of scores for each subject.

• Display the results as a DataFrame.

Exercise 2: Filtering Students

• Select students whose age is over 20 years old.

• Display the names and ages of these students.

Exercise 3: Ranking Students

• Calculate the students' grade point average for each subject using the coefficients.

• Rank students based on their grade point average from highest to lowest.

• Display the top 10 students in the ranking with their names and grade point averages.

Exercise 4: Viewing Notes

• Create a bar graph representing the average grade for each subject.

• Add axis labels and a title to the chart.

Exercise 5: Grouping by Age

• Group students by age and calculate the average grades for each age group.

• Display the results as a DataFrame.

Exercise 6: Data Export

• Save the DataFrame containing students over 20 years old in a CSV file called "students_over_20_ans.csv".

Exercise 7: Filtering by Material

• Select students who obtained a score above 15 in Mathematics.

• View their names, ages and grades in Mathematics.

Exercise 8: Data Modification

• Increase the Chemistry grades of all students by 1 point.

• Save changes to the original CSV file.

Exercise 9: Correlation Analysis

• Calculate the correlation between grades in Mathematics and grades in Physics.

• Display the result.

Exercise 10: Exploring Coefficients

• Identify the material with the highest coefficient.

• Display the names of students who chose this subject and their associated coefficient.

4 Statistical Analysis

4.1 Introduction to Statistical Analysis

4.1.1 Presentation of the basic concepts of statistical analysis.

Statistical analysis is a discipline that aims to explore, summarize and interpret data to derive meaningful information from it. Here are some basic concepts of statistical analysis:

Population and sample:

• **Population:** This is the complete set of all observations or data that interest us in a study. For example, if you are doing a study on the size of all students of a school, the population would be the set of all the students of this school.

• **Sample:** In general, it is difficult or expensive to collect data on an entire population. Therefore, you can collect data on a subset of the population called a sample. Statistical analysis is often based on samples representative.

Variables:

• **Dependent variable:** This is the variable that you measure or study in your analysis. It is sometimes called the response variable. For example, if you are studying the impact of a drug on blood pressure, blood pressure would be the dependent variable.

• **Independent variable:** This is the variable that you manipulate or study as a factor influencing the dependent variable. In the previous example, the drug would be the variable independent.

Descriptive Statistics: Descriptive statistics are used to summarize and describe the main characteristics of the data. Common descriptive measures include mean, median, standard deviation, variance, minimum, maximum and quartile.

Probability distribution: A probability distribution describes how values of a variable are distributed. Common distributions include the normal distribution (shaped bell), uniform distribution (all values have the same probability), and many others.

Inferential Statistics: Inferential statistical analysis allows draw general conclusions or make predictions about a population based on a sample. This includes parameter estimation, hypothesis testing and intervals of trust.

Hypothesis testing: Hypothesis testing is used to evaluate whether a given hypothesis on a population is true or not. They generally compare the sample data to a null hypothesis and make it possible to determine whether the observed differences are statistically significant.

Confidence intervals: Confidence intervals are used to estimate the likely range of values of a population parameter. For example, we can estimate the confidence interval 95% for the population average.

Regression: Regression analysis makes it possible to model the relationship between one or more independent variables and one dependent variable. It is commonly used for prediction and the explanation of phenomena.

Non-parametric tests: In addition to parametric tests (based on assumptions about the distribution of data), there are non-parametric tests which do not make assumptions about data distribution. They are used when the conditions for parametric tests are not not fulfilled.

Data Visualization: Charts and visualizations are essential for exploring and present the data in an understandable manner. Common chart types include histograms, box plots, bar charts and scatter plots. Statistical analysis is a broad and complex discipline, and these basic concepts provide the foundation on which the understanding of more advanced statistical methods is based. She is widely used in many fields, including science, economics, medicine, sociology, psychology and many others, to make informed decisions and obtain information to from the data.

4.1.2 Types of statistical data: qualitative and quantitative variables.

In statistics, data can be classified into two main types: qualitative variables and quantitative variables. These two types of variables are used to describe and analyze different aspects of the data.

Qualitative (or categorical) variables: Qualitative variables, also called categorical variables, represent non-numerical characteristics or categories. They can be divided into two subtypes:

• **Nominal variables:** These variables represent categories or labels without intrinsic order. For example, the color of cars (red, blue, green) is a variable nominal, because there is no notion of order between the colors.

• **Ordinal variables:** These variables represent categories with a natural order. By example, the level of education (primary, secondary, higher) is an ordinal variable, because there is a logical order between the levels of education.

Common statistical operations for qualitative variables include calculating the frequency (number of occurrences of each category) and the creation of contingency tables to analyze the relationships between two qualitative variables.

Quantitative variables: Quantitative variables are numerical variables that represent measurements or quantities. They can be divided into two subtypes:

• **Continuous variables:** These variables can take any value in a given interval. For example, height, weight, body temperature are variables continuous.

• **Discrete variables:** These variables can only take specific and distinct values. For example, the number of people in a family is a discrete variable, because it can only be 0, 1, 2, 3, etc.

Common statistical operations for quantitative variables include calculating measures of central tendency (mean, median, mode), measure of dispersion (standard deviation, variance), creating histograms and performing statistical tests such as regression and analysis of variance.

It is important to correctly distinguish qualitative variables from quantitative variables during statistical analysis, because analysis methods and appropriate visualizations differ depending on the data type.

Poor classification of variables can lead to erroneous conclusions or inappropriate interpretation of results.

4.1.3 Examples of problems that can be solved with statistical analysis.

Statistical analysis can be used to solve a wide variety of problems in many domains, and Python, with its libraries such as NumPy, Pandas, Matplotlib and SciPy, is a popular choice for performing these analyses. Here are some examples of problems which you can solve using statistical analysis with Python:

Sales Analysis: You can use statistical analysis to study sales trends. sales, identify the most popular products, predict future sales and segment customers based on their purchasing behavior.

Market research: You can analyze survey data to understand the consumer preferences, purchasing habits, customer satisfaction, etc.

Financial analysis: Statistical analysis can help evaluate financial performance, calculate returns, estimate risks and construct investment portfolios diversified.

Epidemiology: In the field of public health, statistical analysis is used to study the spread of diseases, evaluate the effectiveness of treatments and health interventions, and model epidemic trends.

Quality Control: Statistical analysis is used to monitor product quality in factories using methods such as control charts and acceptance testing.

Weather forecasting: Statistical models can be used to forecast weather forecasts. Future weather conditions by analyzing historical weather data.

Text and natural language analysis: You can use statistical analysis to extract useful information from texts, such as sentiment analysis, categorization documents, etc.

Scientific studies: In science, statistical analysis is essential for the analysis of experiments and laboratory data, as well as for modeling natural phenomena.

Digital Marketing: Statistical analysis of online data can help understand the user behavior, optimize advertising campaigns and improve usability websites.

Economic Forecasting: You can use econometric models to forecast the economic trends, unemployment, inflation, etc.

To perform these analyzes with Python, you can use libraries like Pandas to data manipulation, Matplotlib and Seaborn for visualization, NumPy for calculations numerical, and SciPy for more advanced statistics. Additionally, there are domain-specific libraries that can be used in conjunction with Python for analytics more specialized.

4.2 Using NumPy and Pandas for Data Analysis

NumPy and Pandas are two popular Python libraries for data analysis. NumPy is mainly used to perform efficient numerical calculations, while Pandas is specially designed for handling tabular data. This is how you can use these two libraries for data analysis:

4.2.1 Using NumPy:

Creating NumPy Arrays: You can create NumPy arrays to store digital data. For example :

```
import numpy as np

data = np.array([1, 2, 3, 4, 5])
```

Numerical operations: NumPy offers a wide variety of efficient numerical operations, such as calculating mean, median, variance, etc. :

```
mean = np.mean(data)

median = np.median(data)

variance = np.var(data)
```

Array Operations: You can perform operations on NumPy arrays, such as cutting, concatenation, transposing, etc. :

```
subarray = data[1:4]

concatenate = np.concatenate([data, [6, 7, 8]])

transposed = np.transpose(data)
```

4.2.2 Use of Pandas:

Creating DataFrames: Pandas DataFrames are useful for storing and manipulating data tabular data. You can create a DataFrame from lists, dictionaries or by importing data from CSV, Excel, etc files :

```
import pandas as pd

data = {
    'Name': ['Alice', 'Bob', 'Charlie'],
    'Age': [25, 30, 35],
    'City': ['Paris', 'New York', 'London']
}

df = pd.DataFrame(data)
```

Data manipulation: Pandas offers many features to manipulate and analyze tabular data. You can perform operations such as filtering, sorting, grouping, column selection, etc. :

filtered_df = df[df['Age'] > 25]

sorted_df = df.sort_values(by='Age')

grouped_df = df.groupby('City')['Age'].mean()

Importing and exporting data: Pandas supports importing data to from files such as CSV, Excel, SQL, etc., and exporting data to different formats:

imported_df = pd.read_csv('data.csv')

df.to_csv('new_data.csv', index=False)

Data Visualization: Although Pandas is not primarily a data visualization library, visualization, it can be used in conjunction with visualization libraries like Matplotlib and Seaborn to create graphs and charts from data in a DataFrame.

import matplotlib.pyplot as plt

df['Age'].plot(kind='hist')

plt.show()

NumPy and Pandas are essential libraries for data analysis in Python. NumPy is ideal for numerical calculations, while Pandas is specifically designed for manipulating and analyze tabular data, making them powerful tools for exploration and analysis of data.

4.3 Practical case of statistical analysis with Python

4.3.1 Importing the necessary libraries:

First, we import the necessary libraries including Pandas and Matplotlib to data manipulation and visualization.

[46]:
```
import pandas as pd

import matplotlib.pyplot as plt
```

4.3.2 Loading data: We use Pandas to load data from a CSV file. Suppose the file CSV contains two columns: “Month” and “Sales”.

[48]:
```
# Loading data from a CSV file

data = pd.read_csv('monthly_sales.csv')
```

4.3.3 Data exploration:

Before performing statistical analyses, it is important to understand the data. We We can display the first few rows of the DataFrame to get a preview of the data.

[49]:
```
print(data.head())
```

Month Sales

0 Jan 4263

1 Feb 3445

2 Mar 1806

3 Apr 4057 4 May 2432

4.3.4 Statistical summary:

We can obtain a statistical summary of monthly sales using the method Pandas describe().

[50]:
```
summary_stats = data['Sales'].describe()
    print(summary_stats)
```

count 12.000000

mean 3038.916667

std 1204.999921

min 1038.000000

25% 2275.500000

50% 3474.500000

75% 3987.250000

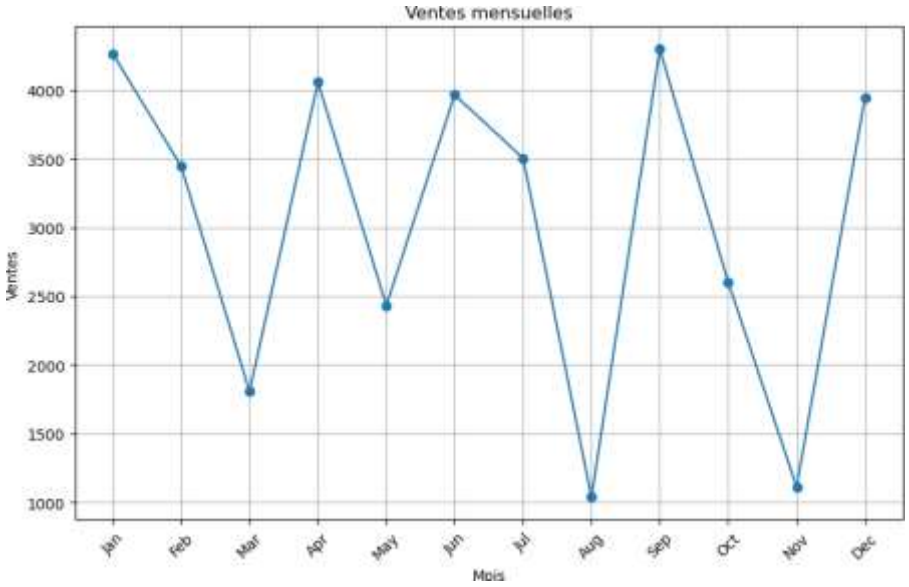
max 4297.000000

Name: Sales, dtype: float64

4.3.5 Data visualization:**

We can create a line chart to visualize monthly sales over time.

```
[51]: plt.figure(figsize=(10, 6))
plt.plot(data['Month'], data['Sales'], marker='o', linestyle='-')
plt.xlabel('Month')
plt.ylabel('Sales')
plt.title('Monthly sales')
plt.xticks(rotation=45)
plt.grid(True)
plt.show()
```



4.3.6 Trend analysis:

We can average monthly sales to get an idea of the trend general.

```
[53]: average_sales = data['Sales'].mean()
print(f"The average monthly sales is: {average_sales}")
```

The average monthly sales is: 3038.9166666666665

This simple example illustrates how to perform basic statistical analysis with Python by using Pandas for data manipulation and Matplotlib for visualization. Of course, In real cases, statistical analysis can become much more complex depending on the nature of the data and questions asked.

5 Practical Work

5.1 TP 1: Data exploration

Objective: Learn how to explore a dataset and get a basic overview.

1. Upload a real dataset or create a dummy dataset.

2. Load the data into a Pandas DataFrame.

3. View the first few rows of the DataFrame to understand the data structure.

4. Calculate descriptive statistics such as mean, median, standard deviation, etc.

5. Create charts to visualize key features of the data, such as histograms, box plots, etc.

5.2 TP 2: Trend analysis

Objective: Learn to analyze the trend of data over time.

1. Use a time series dataset (for example, monthly sales of a product).

2. Draw a line graph to visualize the time trend.

3. Calculate the moving average over multiple periods to smooth the data.

4. Identify possible trends and seasons in the data.

5.3 TP 3: Hypothesis testing

Objective: Learn how to perform a statistical hypothesis test.

1. Formulate a null hypothesis and an alternative hypothesis about a data set (for example, a hypothesis on the difference between two groups).

2. Choose an appropriate statistical test (t-test, ANOVA test, chi-square test, etc.).

3. Perform the test and interpret the results.

4. Determine whether you can accept or reject the null hypothesis.

5.4 TP 4: Linear regression

Objective: Learn how to perform linear regression analysis.

1. Choose a dataset with a dependent (target) variable and one or more variables independent (predictors).

2. Use linear regression to model the relationship between the dependent variable and the independent variables.

3. Interpret the regression coefficients and model quality.

4. Make predictions using the regression model.

5.5 TP 5: Analysis of multidimensional data

Objective: Learn to explore and analyze multidimensional data.

1. Use a dataset with multiple variables.

2. Use multivariate analysis methods such as principal component analysis (PCA) or multiple correspondence analysis (MCA).

3. Visualize the results using two- or three-dimensional graphs to explore the relationships between variables.

6 Advanced Practical Work

6.1 TP 1: Multiple regression analysis

Objective: Perform multiple regression analysis to model complex relationships between several variables.

1. Choose a dataset with one dependent variable and several independent variables.

2. Perform multiple regression analysis to model the relationship between the dependent variable and the independent variables.

3. Evaluate model quality using metrics such as coefficient of determination (R^2) and cross-validation.

6.2 TP 2: Advanced time series analysis

Objective: Use more advanced techniques for time series analysis.

1. Choose a time series dataset and perform exploratory analysis advanced.

2. Apply advanced time series models such as ARIMA or SARIMA models for forecasting.

3. Evaluate model performance using metrics such as the square root of the root mean square error (RMSE).

6.3 TP 3: Analysis of geospatial data

Objective: Perform statistical analysis of geospatial data.

1. Use a geospatial dataset containing geographic coordinates.

2. Create maps to visualize the spatial distribution of data.

3. Perform spatial clustering analysis to identify spatial groupings.

4. Use mapping tools to create thematic maps.

6.4 TP 4: Logistic regression analysis

Objective: Perform logistic regression analysis for modeling binary data.

1. Choose a dataset with a binary variable as the target variable.

2. Perform logistic regression analysis to model the relationship between the target variable and explanatory variables.

3. Interpret the logistic regression coefficients and assess the quality of the model using metrics such as AUC-ROC.

6.5 TP 5: Analysis of textual data

Objective: Perform statistical analysis of textual data.

1. Use a text dataset (for example, customer reviews).

2. Perform text analysis to extract information such as the most popular keywords frequent, feelings, etc.

3. Create visualizations such as word clouds and sentiment charts.

6.6 TP 6: Survival analysis

Objective: Perform a survival analysis to study the time to an event.

1. Choose a survival dataset that includes information on time to a event (for example, patient survival time).

2. Use survival analysis methods such as the Kaplan-Meier model or the by Cox. 3. Identify factors that influence survival and interpret the results. These more advanced practicals will allow you to explore specific areas of statistical analysis, such as multiple regression, advanced time series, geospatial data, regression logistics, text analysis and survival analysis, while deepening your skills in Python and statistics.

Conclusion

Programming with Python software is a process that requires mastery of codes and different commands. The Python language allows you to write dense programs, which contributes to its adoption by several institutions such as the NASA Station.

To successfully program, you must have a good understanding of the operating system you are using. Algorithms also facilitate the implementation of the program architecture to be established to solve a problem in any domain.