



INSTITUTE OF AERONAUTICAL ENGINEERING

(Autonomous)

Dundigal - 500 043, Hyderabad, Telangana

COURSE CONTENT

DATA HANDLING AND VISUALIZATION LABORATORY								
IV Semester: CSE (DS)								
Course Code	Category	Hours / Week			Credits	Maximum Marks		
		L	T	P		C	CIA	SEE
ACDD04	Core	0	0	2	1	40	60	100
Contact Classes: Nil	Tutorial Classes: Nil	Practical Classes: 45				Total Classes: 45		
Prerequisites: Python Programming								

I. COURSE OVERVIEW:

Data handling is the process of collecting, organizing, and presenting the data in a way to analyse, make predictions, draw conclusions, and make decisions. Data visualization is a part of exploratory data analysis, a prior step before a full-pledged data analysis. This laboratory course is intended to offer practical knowledge and skills in both data handling and visualization. In this laboratory, python packages such as NumPy, SciPy and Pandas used for computations, and the visualization packages such as seaborn and matplotlib are practiced. Hands-on exercises are designed to explore the basic data importing, exploration, visualization, preliminary data analysis and data exporting techniques using core python and its packages. The expertise gained in this laboratory lays foundation for detailed data analysis that involves data modelling, analysis, evaluation and mining in scientific and engineering domains.

II. COURSE OBJECTIVES

The students will try to learn:

- I. Installation and usage of python packages useful for data exploration and visualization.
- II. Data handling using python in practice.
- III. The practical knowledge of data visualization capabilities of python packages.

III. COURSE OUTCOMES

At the end of the course students should be able to:

- CO 1 Tabulate the data from the CSV, XLS, TXT and JSON files as data frames and export the data frame to files.
- CO 2 Make use of imputation techniques for wrangling the data using pandas package.
- CO 3 Create the python dataframes to form pivot tables and contingency tables.
- CO 4 Manipulate the tabular data by joining multiple dataframes using pandas package.
- CO 5 Explore the data using the data visualization techniques in python environment.
- CO 6 Analyze the data for outliers to data trimming the data required for an authentic data analysis in python environment.

IV. COURSE CONTENT

DATA HANDLING AND VISUALIZATION LABORATORY (ACDD04) CONTENTS

S No.	Topic Name	Page No.
1	Installation of python and related packages	4-18
	a. Install python, and packages; NumPy, SciPy and Panda.	
	b. Study matrix operations: rank, inverse, condition number	
	c. Solving for simultaneous equations in 3 or 4 variables.	
2	Working with CSV files and XLS files.	19-32
	a. Save a List to CSV, XLSX and TXT files.	
	b. Save a Dictionary to CSV, XLSX and TXT files.	
	c. Load data from CSV, XLSX and TXT pandas to a List.	
	d. Load data from CSV, XLSX and TXT pandas to a Dictionary.	
3	Basic operations on Dataframe.	33-36
	a. Attribute filtering based on conditions.	
	b. Attribute filtering based on slicing.	
	c. Attribute filtering based on queries.	
4	Summary Statistics of the data	37-46
	a. Compute ranking statistics of the data.	
	b. Compute statistical averages of numerical attributes.	
	c. Compute statistical ratios of numerical attributes.	
	d. Interpret the results.	
5	Handling Missing Values	47-58
	a. Drop the rows containing missing values	
	b. Impute missing values with statistical averages.	
	c. Impute missing values using linear interpolation.	
	d. Interpret the results.	
6	Handling Time series data.	59-69
	a. Display the date and time information in different formats.	
	b. Generate summary statistics during a period.	
	c. Compute rolling mean and rolling std deviations and plot.	
7	Visualization of categorical data	70-79
	a. Plot categorical data as vertical and horizontal bar charts and label it.	
	b. Plot categorical data as vertical grouped bar chart and label it.	
	c. Plot categorical data as vertical stacked bar chart and label it.	
	d. Interpret the results.	
8	Visualization of correlations.	80-91
	a. Plot the pair wise scatter plots of numerical attributes	
	b. Identify the type of correlations.	
	c. Interpret the results.	
9	Visualization of distributions	92-96
	a. Plot the histograms of numerical data.	

	b. Plot the counts of categorial data.	
	c. Plot the data distributions (or densities).	
	d. Interpret the results.	
10	Visualization using box-and-whisker plots.	97-102
	a. Compute the rank statistics of numerical attributes.	
	b. Create the box-and-whisker plots of numerical attributes.	
	c. Interpret the results.	
11	Handling outliers in the data.	103-109
	a. Identify the outliers using quartile method.	
	b. Identify the outliers using standard deviation method.	
	c. Compare the performance of two methods.	
	d. Remove outliers from the data.	
	e. Interpret the results.	
12	Working with Data Tables.	110-112
	a. Joining the data tables.	
	b. Exercises on contingency tables	
	c. Exercises on grouping data.	
13	Data Scaling and Transformation.	113-118
	a. Scaling the data using different python scalers.	
	b. Normalization as a special case of data scaling.	
	c. Data transformation using standardization.	
	d. Compare the results and interpret.	
14	Web Scraping.	119-126
	a. Scraping a list of items from a website.	
	b. Scraping data from a table.	
	c. Scraping images from a website.	
	d. Scraping data with pagination.	

V. SYLLABUS:

EXERCISES FOR DATA HANDLING AND VISUALIZATION LABORATORY

Note: Students are encouraged to bring their own laptops for laboratory practice sessions.

Getting Started Exercises

1. Installation of python and related packages

a. Install python, and packages: NumPy, Panda and SciPy.

To install Python and the packages NumPy, SciPy, and Pandas, follow these steps:

Install Python:

To install Python on a Windows system, you can:

Step 1: Select Version to Install Python

Visit the official page for Python <https://www.python.org/downloads/> on the Windows operating system. Locate a reliable version of Python 3, preferably version 3.10.11, which was used in testing this tutorial. Choose the correct link for your device from the options provided: either **Windows installer (64-bit)** or **Windows installer (32-bit)** and proceed to download the executable file.

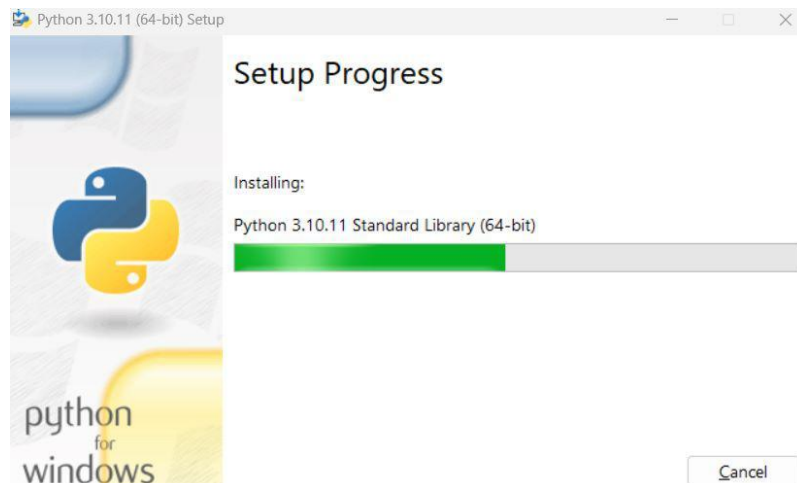


Step 2: Downloading the Python Installer

Once you have downloaded the installer, open the .exe file, such as **python-3.10.11-amd64.exe**, by double-clicking it to launch the Python installer. Choose the option to Install the launcher for all users by checking the corresponding checkbox, so that all users of the computer can access the Python launcher application. Enable users to run Python from the command line by checking the Add python.exe to PATH checkbox.

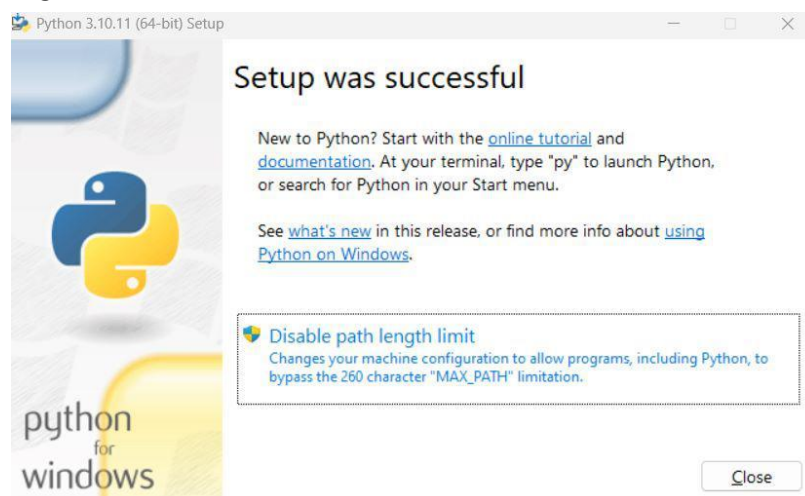


After Clicking the **Install Now Button** the setup will start installing Python on your Windows system. You will see a window like this.



Step 3: Running the Executable Installer

After completing the setup. Python will be installed on your Windows system. You will see a successful message.

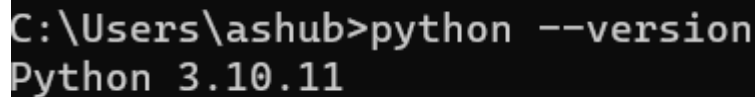


Step 4: Verify the Python Installation in Windows

Close the window after successful installation of Python. You can check if the installation of Python

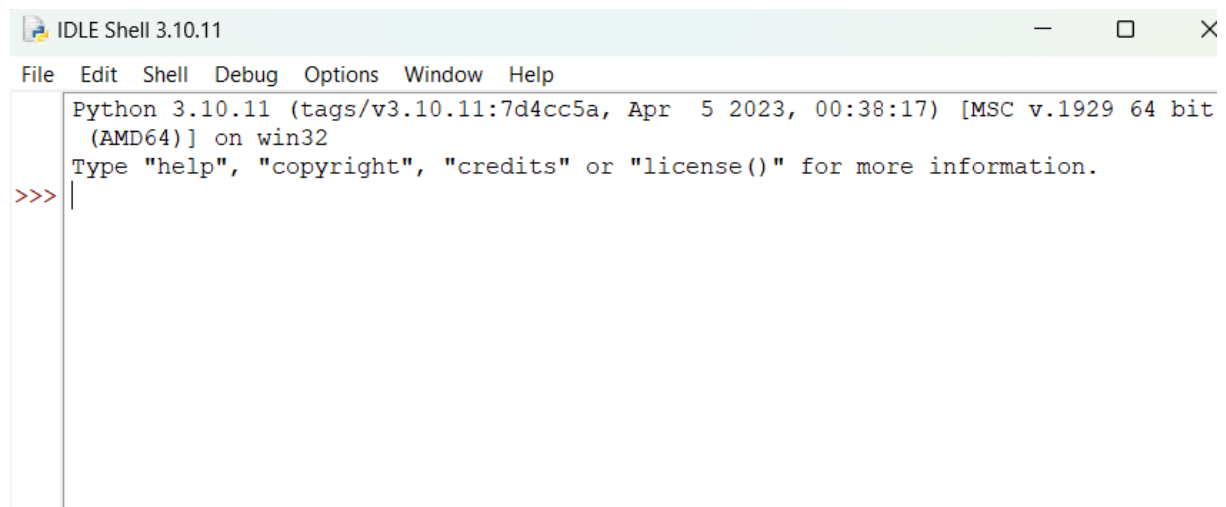
was successful by using either the command line or the **Integrated Development Environment (IDLE)**, which you may have installed. To access the command line, click on the Start menu and type "cmd" in the search bar. Then click on Command Prompt.

```
python --version
```



```
C:\Users\ashub>python --version
Python 3.10.11
```

You can also check the version of Python by opening the IDLE application. Go to Start and enter IDLE in the search bar and then click the IDLE app, for example, **IDLE (Python 3.10.11 64-bit)**. If you can see the Python IDLE window then you are successfully able to download and installed Python on Windows.



Install Packages:

You can list installed Python packages by using the pip command-line tool with the list command.

1. Installing Numpy on Windows:

Python NumPy is a general-purpose array processing package that provides tools for handling **n-dimensional arrays**. It provides various computing tools such as comprehensive mathematical functions, and linear algebra routines. NumPy provides both the flexibility of **Python** and the speed of well-optimized compiled C code. Its easy-to-use syntax makes it highly accessible and productive for programmers from any background. In this article, we will see how to install NumPy as well as how to import Numpy in Python.

Pre-requisites:

- Knowledge on *Python libraries*
- Anaconda*
- Pycharm*

Installing Numpy For PIP Users

Users who prefer to use pip can use the below command to install NumPy:

```
pip install numpy
```

You will get a similar message once the installation is complete:

```
C:\ Command Prompt
Microsoft Windows [Version 10.0.19042.1202]
(c) Microsoft Corporation. All rights reserved.

C:\Users\Geeks>pip install numpy
Collecting numpy
  Downloading numpy-1.21.2-cp39-cp39-win_amd64.whl (14.0 MB)
    |#####| 14.0 MB 6.4 MB/s
Installing collected packages: numpy
Successfully installed numpy-1.21.2
```

Install Numpy Using Conda

If you want the installation to be done through conda, you can use the below command:

```
conda install -c anaconda numpy
```

You will get a similar message once the installation is complete

```
Anaconda Powershell Prompt (anaconda3)
The following packages will be downloaded:

package | build |
-----|-----|
blas-1.0 | mkl | 6 KB | anaconda
-----|-----|
Total: 6 KB

The following NEW packages will be INSTALLED:

blas          anaconda/win-64::blas-1.0-mkl
intel-openmp  pkgs/main/win-64::intel-openmp-2021.3.0-haa95532_3372
mkl           pkgs/main/win-64::mkl-2021.3.0-haa95532_524
mkl-service   pkgs/main/win-64::mkl-service-2.4.0-py38h2bbff1b_0
mkl_fft       pkgs/main/win-64::mkl_fft-1.3.0-py38h277e83a_2
mkl_random    pkgs/main/win-64::mkl_random-1.2.2-py38hf11a4ad_0
numpy         pkgs/main/win-64::numpy-1.20.3-py38ha4e8547_0
numpy-base    pkgs/main/win-64::numpy-base-1.20.3-py38hc2deb75_0

Proceed ([y]/n)? y

Downloading and Extracting Packages
blas-1.0 | 6 KB | ##### | 100%
Preparing transaction: done
Verifying transaction: done
Executing transaction: done
```

2. Install Pandas on Windows

Python Pandas can be installed on Windows in two ways:

Using pip

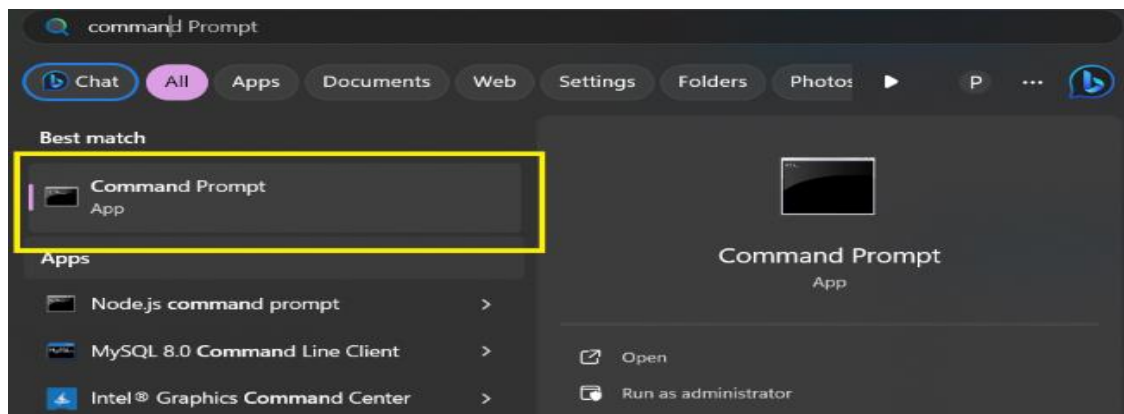
Using Anaconda

Install Pandas using pip

pip is a package management system used to install and manage software packages/libraries written in Python. These files are stored in a large "online repository" termed as Python Package Index (PyPI).

Step 1 : Launch Command Prompt

To open the Start menu, press the Windows key or click the Start button. To access the Command Prompt, type "cmd" in the search bar, click the displayed app, or use Windows key + R, enter "cmd," and press Enter.



Step 2 : Run the Command

Pandas can be installed using PIP by use of the following command in Command Prompt.

```
pip install pandas
```

```
Microsoft Windows [Version 10.0.22621.2715]
(c) Microsoft Corporation. All rights reserved.

C:\Users\GFG0371>pip install pandas
Defaulting to user installation because normal site-packages is not writeable
Collecting pandas
  Downloading pandas-2.1.3-cp312-cp312-win_amd64.whl.metadata (18 kB)
Requirement already satisfied: numpy<2, >=1.26.0 in c:\users\gfg0371\appdata\roaming\python\python312\site-packages (from pandas)
Requirement already satisfied: python-dateutil>=2.8.2 in c:\users\gfg0371\appdata\roaming\python\python312\site-packages (from pandas)
Collecting pytz>=2020.1 (from pandas)
  Downloading pytz-2023.3.post1-py2.py3-none-any.whl.metadata (22 kB)
Collecting tzdata>=2022.1 (from pandas)
  Downloading tzdata-2023.3-py2.py3-none-any.whl (341 kB)
 341.8/341.8 kB 1.5 MB/s eta 0:00:00
Requirement already satisfied: six>=1.5 in c:\users\gfg0371\appdata\roaming\python\python312\site-packages (from pandas)
Downloading pandas-2.1.3-cp312-cp312-win_amd64.whl (10.5 MB)
 10.5/10.5 MB 2.1 MB/s eta 0:00:00
Downloading pytz-2023.3.post1-py2.py3-none-any.whl (502 kB)
 502.5/502.5 kB 3.2 MB/s eta 0:00:00
Installing collected packages: pytz, tzdata, pandas
```

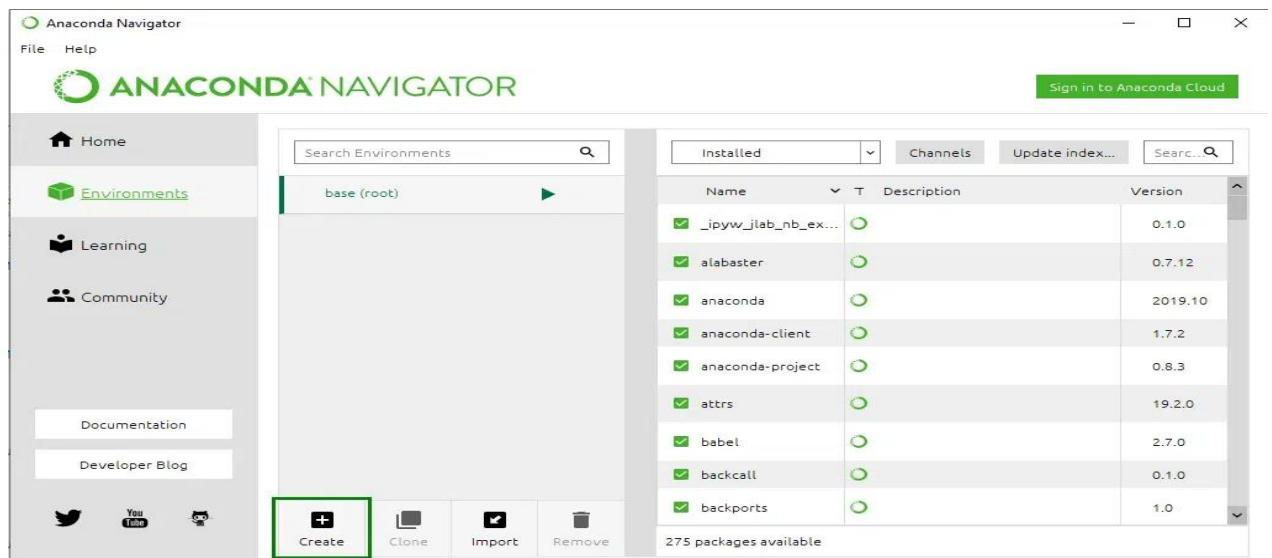
Install Pandas using Anaconda

Anaconda is open-source software that contains Jupyter, spyder, etc that is used for large data processing, Data Analytics, and heavy scientific computing. If your system is not pre-equipped with Anaconda Navigator, you can learn **how to install Anaconda Navigator on Windows or Linux**.

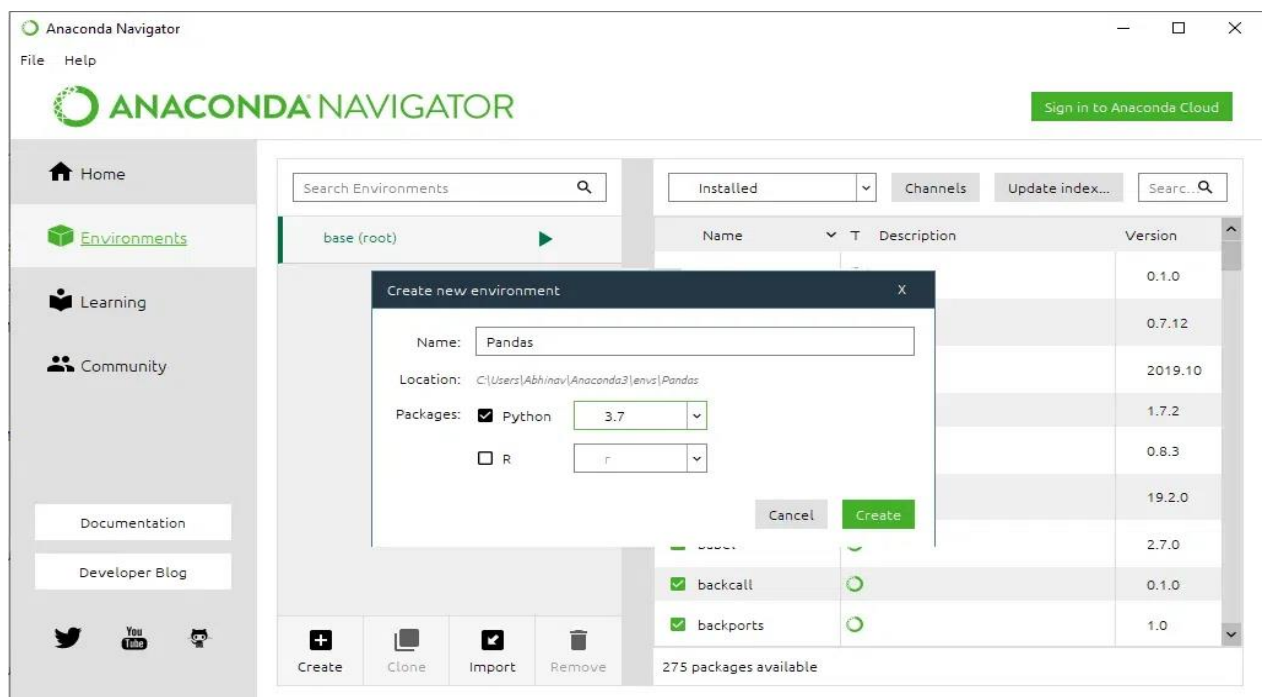
Install and Run Pandas from Anaconda Navigator

Step 1: Search for **Anaconda Navigator** in Start Menu and open it.

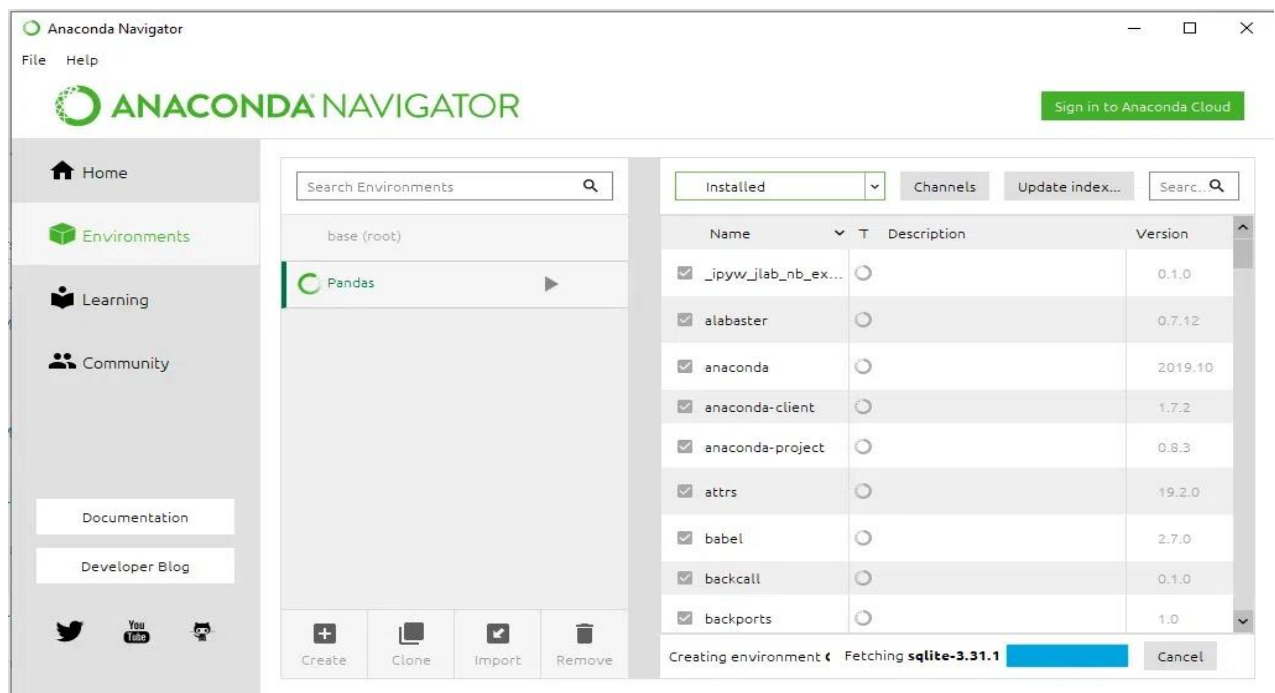
Step 2: Click on the **Environment tab** and then click on the **Create** button to create a new Pandas Environment.



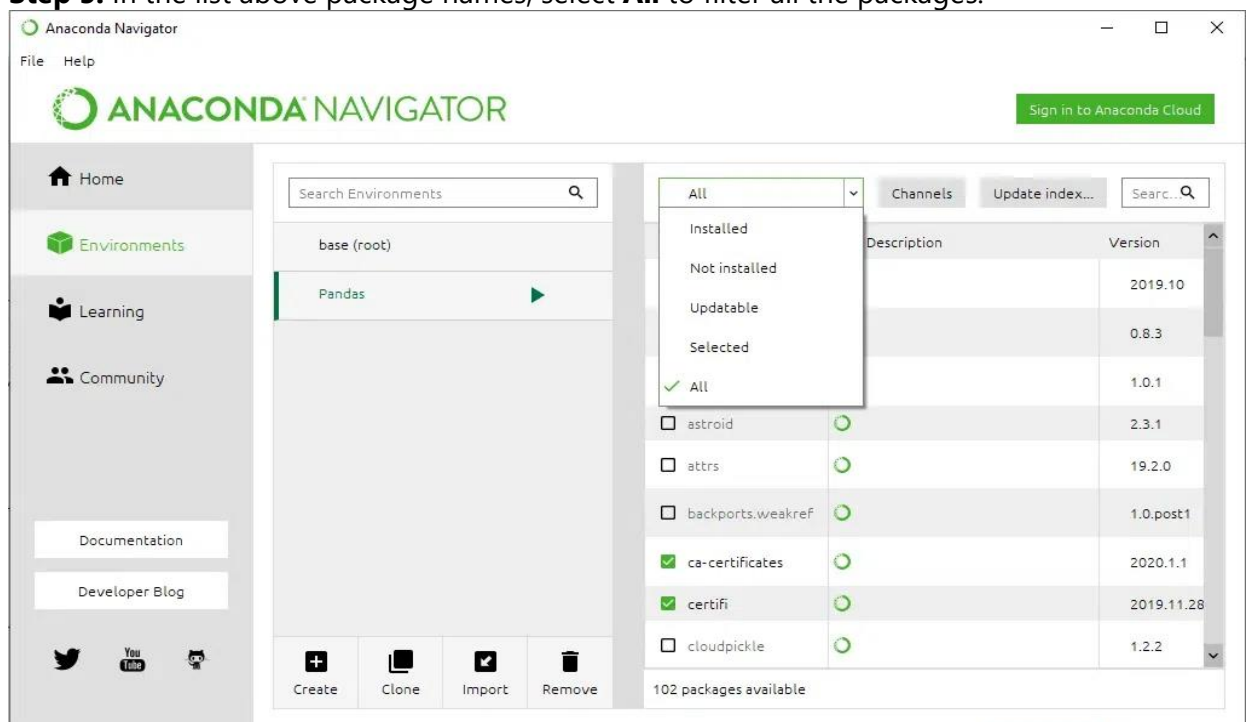
Step 3: Give a name to your Environment, e.g. Pandas, and then choose a Python and its version to run in the environment. Now click on the **Create** button to create Pandas Environment.



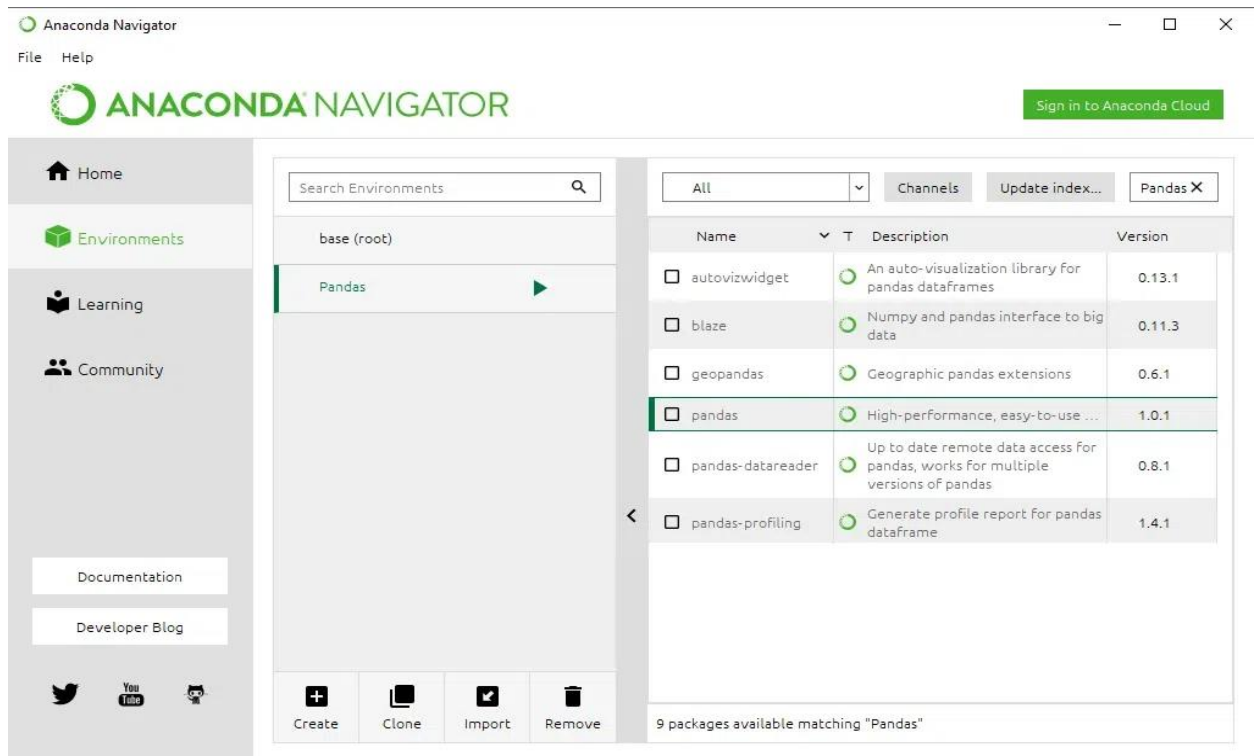
Step 4: Now click on the **Pandas Environment** created to activate it.



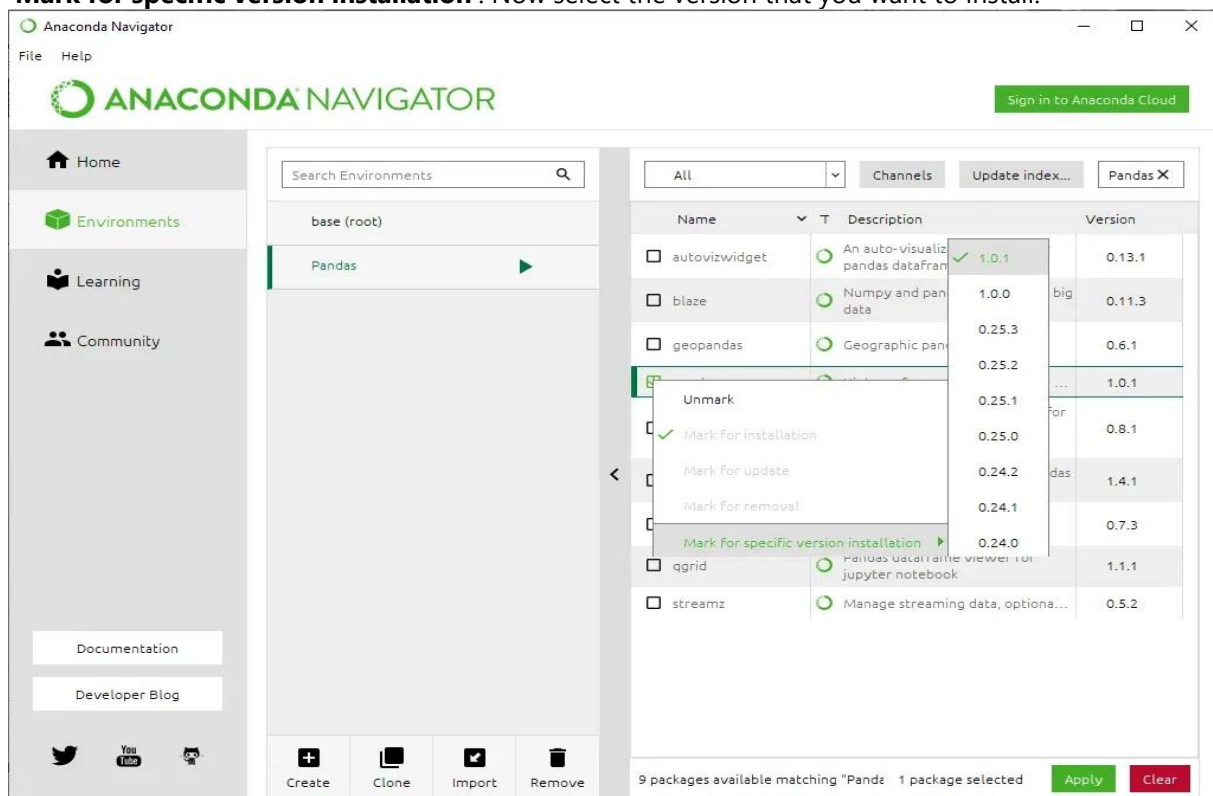
Step 5: In the list above package names, select **All** to filter all the packages.



Step 6: Now in the Search Bar, look for '**Pandas**'. Select the **Pandas package** for Installation.



Step 7: Now Right Click on the checkbox given before the name of the package and then go to **'Mark for specific version installation'**. Now select the version that you want to install.



Step 8: Click on the **Apply** button to install the Pandas Package.

Step 9: Finish the Installation process by clicking on the **Apply** button.

Step 10: Now to open the Pandas Environment, click on the **Green Arrow** on the right of the package name and select the Console with which you want to begin your Pandas programming.

```
C:\Windows\system32\cmd.exe
(Pandas) C:\Users\Abhinav>
```

3. Install Scipy in Python on Windows

For PIP Users:

Users who prefer to use pip can use the below command to install Scipy package on Windows:

```
pip install scipy
```

message once the installation is complete:

```
C:\Users\Geeks>pip install scipy
collecting scipy
  Downloading scipy-1.7.1-cp38-cp38-win_amd64.whl (33.7 MB)
    |#####| 33.7 MB 6.8 MB/s
Requirement already satisfied: numpy<1.23.0,>=1.16.5 in c:\users\geeks\anaconda3\lib\site-packages (from scipy) (1.20.3)
Installing collected packages: scipy
ERROR: After October 2020 you may experience errors when installing or updating packages. This is because pip will change
We recommend you use --use-feature=2020-resolver to test your packages with the new resolver before it becomes the default
seaborn 0.11.2 requires pandas>=0.23, which is not installed.
phik 0.12.0 requires joblib>=0.14.1, which is not installed.
phik 0.12.0 requires pandas>=0.25.1, which is not installed.
pandas-profiling 3.0.0 requires joblib, which is not installed.
pandas-profiling 3.0.0 requires pandas!=1.0.0,!1.0.1,!1.0.2,!1.1.0,>=0.25.3, which is not installed.
imagehash 4.2.1 requires PyWavelets, which is not installed.
Successfully installed scipy-1.7.1
WARNING: You are using pip version 21.0.1; however, version 21.2.4 is available.
You should consider upgrading via the 'C:\Users\Geeks\anaconda3\python.exe -m pip install --upgrade pip' command.
```

Verifying Scipy Module Installation:

To verify if Scipy has been successfully installed in your system run the below code in a python IDE of your choice:

```
import scipy
scipy.__version__
```

If successfully installed you will get the following output.

```
C:\Users\Geeks>python
Python 3.8.5 (default, Sep 3 2020, 21:29:08) [MSC v.1916 64 bit (AMD64)] :: Anaconda, Inc. on win32

Warning:
This Python interpreter is in a conda environment, but the environment has
not been activated. Libraries may fail to load. To activate this environment
please see https://conda.io/activation

Type "help", "copyright", "credits" or "license" for more information.
>>> import scipy
>>> scipy.__version__
'1.7.1'
>>>
```

For Conda Users:

If you want the installation to be done through conda, you can use the below command:

```
conda install scipy
```

Type y for yes when prompted.

You will get a similar message once the installation is complete

```
(base) PS C:\Users\Geeks> conda install scipy
Collecting package metadata (current_repodata.json): done
Solving environment: done

## Package Plan ##

  environment location: C:\Users\Geeks\anaconda3

  added / updated specs:
    - scipy

The following NEW packages will be INSTALLED:

  icc_rt                pkgs/main/win-64::icc_rt-2019.0.0-h0cc432a_1
  scipy                 pkgs/main/win-64::scipy-1.7.1-py38hbe87c03_2

Proceed ([y]/n)? y

Preparing transaction: done
Verifying transaction: done
Executing transaction: done
(base) PS C:\Users\Geeks>
```

Verify Installation

- Open a Python interpreter by typing python in your terminal or command prompt.
- Try to import the packages:

```
import numpy as np
import scipy as sp
import pandas as pd
```

If there are no errors, the packages are installed successfully.

Try:

1. Write a program to check whether a Numpy array contains a specified row?

Sample Output:

```
[[ 1  2  3  4  5]
 [ 6  7  8  9 10]
 [11 12 13 14 15]
 [16 17 18 19 20]]
True
False
False
True
```

2. Write a program to get all 2D diagonals of a 3D NumPy array?

Sample Output:

Original 3d array:

```
[[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]
 [12 13 14 15]]
```

```
[[16 17 18 19]
 [20 21 22 23]
 [24 25 26 27]
 [28 29 30 31]]
```

```
[[32 33 34 35]
 [36 37 38 39]
 [40 41 42 43]
 [44 45 46 47]]]
```

2d diagonal array:

```
[[ 0  5 10 15]
 [16 21 26 31]
 [32 37 42 47]]
```

b. Study matrix operations: rank, inverse, condition number

Operation	Python Function	Key Notes
Rank	<code>np.linalg.matrix_rank</code>	Works for any matrix (not just square).
Inverse	<code>np.linalg.inv</code>	Only for square and nonsingular matrices.
Condition Number	<code>np.linalg.cond</code>	Indicates numerical stability of a matrix.

1. Rank of a Matrix

The **rank** of a matrix indicates the number of linearly independent rows or columns. It reflects the "dimension" of the matrix's space. For example:

- A 3×3 matrix of rank 3 is "full rank."
- A rank-deficient matrix has fewer linearly independent rows or columns.

In Python:

The function **`numpy.linalg.matrix_rank`** determines the rank using numerical methods like the **singular value decomposition (SVD)**. SVD decomposes the matrix, and the rank is the count of non-zero singular values.

```
import numpy as np

A = np.array([[1, 2, 3],
              [4, 5, 6],
              [7, 8, 9]])

# Compute rank
rank = np.linalg.matrix_rank(A)
print("Rank of A:", rank)
```

Output:

Rank of A=2 (because the third row is a linear combination of the first two rows).

2. Inverse of a Matrix

Python provides a very easy method to calculate the inverse of a matrix. The function **numpy.linalg.inv()** is available in the NumPy module and is used to compute the inverse matrix in Python.

Syntax: *numpy.linalg.inv(a)*

Example 1: In this example, we will create a 3 by 3 NumPy array matrix and then convert it into an inverse matrix using the `np.linalg.inv()` function.

```
# Import required package
import numpy as np

# Taking a 3 * 3 matrix
A = np.array([[6, 1, 1],
              [4, -2, 5],
              [2, 8, 7]])

# Calculating the inverse of the matrix
print(np.linalg.inv(A))
```

Output:

```
[[ 0.17647059 -0.00326797 -0.02287582]
 [ 0.05882353 -0.13071895  0.08496732]
 [-0.11764706  0.1503268   0.05228758]]
```

3. Condition Number

The **condition number** of a matrix measures how sensitive the solution of a system is to errors in the input. A small condition number indicates stability, while a large one suggests potential numerical instability.

Mathematically:

Condition Number= Largest Singular Value/ Smallest Singular Value

Significance:

- **Low condition number (close to 1):** Well-conditioned matrix.
- **High condition number:** Ill-conditioned matrix; results may be unreliable.

In Python: *Use [numpy.linalg.cond](#) to compute the condition number.*

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

# Compute the condition number
cond_number = np.linalg.cond(C)
print("Condition number of C:", cond_number)
```

Output:

High condition numbers suggest the matrix is close to singular, leading to unstable computations.

Try:

1. Write a program to compute the eigenvalues and eigenvectors of a complex matrix.

Sample Output:

Eigenvalues: [5.56155281+2.26527142j -0.56155281+0.73472858j]

Eigenvectors:

```
[[ 0.48454084+0.j      0.83703486+0.j      ]
 [ 0.87474491+0.j     -0.54713267+0.j     ]]
```

2. Write a program to compute the inverse a matrix using NumPy?

Sample Output:

```
[[ 0.17647059 -0.00326797 -0.02287582]
 [ 0.05882353 -0.13071895  0.08496732]
 [-0.11764706  0.1503268  0.05228758]]
```

c. Solving for simultaneous equations in 3 or 4 variables

To solve simultaneous equations in 3 or 4 variables, you can represent the system as a matrix equation and use numerical methods or analytical techniques.

Matrix Representation

A system of equations can be written in the form:

$A \cdot X = B$

Where:

- A: Coefficient matrix.
- X: Column vector of variables.
- B: Column vector of constants.

For example, the system:

$$\begin{aligned} 2x + y - z &= 8 \\ -3x - y + 2z &= -11 \\ -2x + y + 2z &= -3 \end{aligned}$$

Can be represented as:

$$\begin{bmatrix} 2 & 1 & -1 \\ -3 & -1 & 2 \\ -2 & 1 & 2 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 8 \\ -11 \\ -3 \end{bmatrix}$$

Steps to Solve

1. Write the coefficient matrix A and the constant vector B .
2. Use the matrix equation $X = A^{-1} \cdot B$ to solve for X , provided A is invertible.

Solving in Python

Example: Solving for 3 variables

Solving equation with three variables

Construct the following equations using Eq() and solve then to find the unknown variables

```
# importing library sympy
from sympy import symbols, Eq, solve

# defining symbols used in equations
# or 3 variables
x, y, z = symbols('x,y,z')

# defining equations
```



```

eq1 = Eq((x+y+z), 1)
print("Equation 1:")
print(eq1)

eq2 = Eq((x-y+2*z), 1)
print("Equation 2")
print(eq2)

eq3 = Eq((2*x-y+2*z), 1)
print("Equation 3")

# solving the equation and printing the
# value of unknown variables
print("Values of 3 unknown variable are as follows:")
print(solve((eq1, eq2, eq3), (x, y, z)))

```

Output: Equation 1:

Eq(x + y + z, 1)

Equation 2

Eq(x - y + 2*z, 1)

Equation 3

Values of 3 unknown variable are as follows:

{x: 0, y: 1/3, z: 2/3}

Example: Solving for 4 variables

```

# Define a system with 4 variables
A = np.array([[1, 2, 3, 4],
              [2, 3, 4, 5],
              [3, 4, 5, 6],
              [4, 5, 6, 7]])

B = np.array([10, 15, 20, 25])

# Solve for the variables
X = np.linalg.solve(A, B)
print("Solution (x1, x2, x3, x4):", X)

```

Applications

- 3 Variables: Common in physics and engineering (e.g., circuits, forces).
- 4 Variables: Often used in economics or systems modeling.

Try:

1. Write a program to solve the following simultaneous equations:

$$2x + 3y = -2$$

$$5x + 4y + 2 = 0$$

Sample Output:

The solution of the given simultaneous equation is (2/7, -6/7)

2. Write a program to solve the following simultaneous equations:

$$a^2 - b = 14 \text{ and } 2b - 4 = 12a$$

Sample Output:

The solution of the given simultaneous equation is $a = 8$, $b = 50$ and $a = -5$, $b = 11$.

2. Working with CSV files and XLS files

1. Save a List to CSV, XLSX and TXT files.

1. Save a List to a CSV File

A CSV (Comma Separated Values) is a simple file format, used to store data in a tabular format. CSV file stores tabular data (numbers and text) in plain text. Each line of the file is a data record. Each record consists of one or more fields, separated by commas. The use of the comma as a field separator is the source of the name for this file format. There are various methods to save lists to CSV which we will see in this article.

Example Code:

The code uses the csv module to write data into a CSV file named 'GFG'. It defines the field names as ['Name', 'Branch', 'Year', 'CGPA'] and the data rows as a list of lists. It opens the file in write mode and uses the csv.writer method to write the field names as the first row and then writes the data rows into the file.

```
import csv
# field names
fields = ['Name', 'Branch', 'Year', 'CGPA']

# data rows of csv file
rows = [ ['Nikhil', 'COE', '2', '9.0'],
         ['Sanchit', 'COE', '2', '9.1'],
         ['Aditya', 'IT', '2', '9.3'],
         ['Sagar', 'SE', '1', '9.5'],
         ['Prateek', 'MCE', '3', '7.8'],
         ['Sahil', 'EP', '2', '9.1']]

with open('GFG', 'w') as f:

    # using csv.writer method from CSV package
    write = csv.writer(f)

    write.writerow(fields)
    write.writerows(rows)
```

Output:

	A	B	C	D
1	Name	Branch	Year	CGPA
2	Nikhil	COE	2	9
3	Sanchit	COE	2	9.1
4	Aditya	IT	2	9.3
5	Sagar	SE	1	9.5
6	Prateek	MCE	3	7.8
7	Sahil	EP	2	9.1
8				

2. Saving a List to an XLSX File

XLSX (Excel format) is commonly used for structured spreadsheets.

- **Library Used:** openpyxl (popular Python library for working with Excel files).

- **Data Format:** Each sublist is written as a row into an Excel sheet.
- **Steps:**

Create a new workbook.

Use `Workbook.active` to access the worksheet.

Write rows to the worksheet using the `.append()` method.

Save the workbook with `.save()`.

Example Code:

```
from openpyxl import Workbook

# List of rows
my_list = [{"Name", "Age", "City"}, ["Alice", 30, "New York"], ["Bob", 25, "Los Angeles"]]

# Create a workbook and worksheet
wb = Workbook()
ws = wb.active

# Append each row to the worksheet
for row in my_list:
    ws.append(row)

# Save as Excel file
wb.save("output.xlsx")
```

Output (output.xlsx):

The file can be opened in Excel or any compatible software. The data will appear in a table format.

3. Saving a List to a TXT File

TXT (Plain Text) is a simple text file where data can be formatted as needed.

- **Library Used:** None (uses Python's built-in file I/O).
- **Data Format:** Rows are written line by line, and elements are separated by a delimiter (e.g., tab `\t` or space).

Steps:

1. Open a file in write mode.
2. Loop through the list and write each sublist as a line.
3. Convert elements to strings and join them with a delimiter.
4. Save and close the file.

Example Code:

```
# List of rows
my_list = [{"Name", "Age", "City"}, ["Alice", 30, "New York"], ["Bob", 25, "Los Angeles"]]

# Save as TXT
with open("output.txt", "w") as file:
    for row in my_list:
        file.write("\t".join(map(str, row)) + "\n") # Convert elements to string & join by tab
```

Output (output.txt):

Name	Age	City
Alice	30	New York
Bob	25	Los Angeles

Comparison of File Formats

Format	File Extension	Use Case	Key Feature
CSV	.csv	Import/export table-like data easily.	Universal, supported by most software.
XLSX	.xlsx	Advanced formatting in Excel files.	Spreadsheet software compatibility.
TXT	.txt	Storing plain text data.	Simple and lightweight.

Try:

1. Write a program to save a Nested List with Headers to a CSV File

Sample Output:

```
ID ,Name, Age, City
101,Alice,25,New York
102,Bob,30,Los Angeles
103,Charlie,28,Chicago
104,Diana,35,Houston
```

2. Write a program to save a List with Multiple Sheets to an XLSX File

Sample Output:

Sheet: Products

Product	Price	Stock
Laptop	1200	50
Phone	800	150
Tablet	400	100

Sheet: Employees

Employee	Department	Salary
Alice	HR	60000
Bob	IT	80000
Charlie	Sales	70000

3. Write a program to save a List with Delimiters to a TXT File

Sample Output:

Student	Math	Science	English
Alice	85	90	88
Bob	78	83	80
Charlie	92	88	95

2. Save a Dictionary to CSV, XLSX and TXT files.

Save a Dictionary to a CSV File

CSV (comma-separated values) files are one of the easiest ways to transfer data in form of string especially to any spreadsheet program like Microsoft Excel or Google spreadsheet. In this article, we will see how to save a Python dictionary to a CSV file. Follow the below steps for the same.

1. Import csv module

```
import csv
```

2. Creating list of field names

```
field_names= ['No', 'Company', 'Car Model']
```

3. Creating a list of python dictionaries

```
cars = [  
    {'No': 1, 'Company': 'Ferrari', 'Car Model': '488 GTB'},  
    {'No': 2, 'Company': 'Porsche', 'Car Model': '918 Spyder'},  
    {'No': 3, 'Company': 'Bugatti', 'Car Model': 'La Voiture Noire'},  
    {'No': 4, 'Company': 'Rolls Royce', 'Car Model': 'Phantom'},  
    {'No': 5, 'Company': 'BMW', 'Car Model': 'BMW X7'},  
]
```

4. Writing content of dictionaries to CSV file

```
with open('Names.csv', 'w') as csvfile:  
    writer = csv.DictWriter(csvfile, fieldnames=field_names)  
    writer.writeheader()  
    writer.writerows(cars)
```

Syntax:

```
DictWriter( (filename), fieldnames = [list of field names] )
```

In the above code snippet **writer** is an instance of csv.DictWriter class and uses two of its following methods:

- DictWriter.**writeheader()** is used to write a row of column headings / field names to the given CSV file
- csvwriter.**writerows()** method is used to write rows of data into the specified file.

Note: To write a single dictionary in CSV file use writerow() method

```
import csv  
  
field_names = ['No', 'Company', 'Car Model']  
  
cars = [  
    {'No': 1, 'Company': 'Ferrari', 'Car Model': '488 GTB'},  
    {'No': 2, 'Company': 'Porsche', 'Car Model': '918 Spyder'},  
    {'No': 3, 'Company': 'Bugatti', 'Car Model': 'La Voiture Noire'},  
    {'No': 4, 'Company': 'Rolls Royce', 'Car Model': 'Phantom'},  
    {'No': 5, 'Company': 'BMW', 'Car Model': 'BMW X7'},  
]
```

```

with open('Names.csv', 'w') as csvfile:
    writer = csv.DictWriter(csvfile, fieldnames = field_names)
    writer.writeheader()
    writer.writerows(cars)

```

Output:

	A	B	C
1	No	Company	Car Model
2	1	Ferrari	488 GTB
3	2	Porsche	918 Spyder
4	3	Bugatti	La Voiture Noire
5	4	Rolls Royce	Phantom
6	5	BMW	BMW X7
7			
8			

OR

The code imports the pandas library as pd. It defines three lists: nme for names, deg for degrees, and scr for scores. It creates a dictionary dict using these lists. Then, it creates a pandas DataFrame df from the dictionary. Finally, it saves the DataFrame as a CSV file named 'GFG.csv' using the to_csv method. The resulting CSV file will contain the columns 'name', 'degree', and 'score' with the corresponding data from the lists.

```

# importing pandas as pd
import pandas as pd

# list of name, degree, score
nme = ["aparna", "pankaj", "sudhir", "Geeku"]
deg = ["MBA", "BCA", "M.Tech", "MBA"]
scr = [90, 40, 80, 98]

# dictionary of lists
dict = {'name': nme, 'degree': deg, 'score': scr}

df = pd.DataFrame(dict)

# saving the dataframe
df.to_csv('GFG.csv')

```

Output:

	A	B	C	D
1		name	degree	score
2	0	aparna	MBA	90
3	1	pankaj	BCA	40
4	2	sudhir	M.Tech	80
5	3	Geeku	MBA	98
6				

```

import csv
import pandas as pd

# Sample list
data = [{"Name", "Age", "City"}, [{"John", 25, "New York"}, {"Emma", 28, "London"}]]

```

```

# Save to CSV
with open("data.csv", "w", newline="") as f:
    writer = csv.writer(f)
    writer.writerows(data)

# Save to XLSX
df = pd.DataFrame(data[1:], columns=data[0])
df.to_excel("data.xlsx", index=False)

# Save to TXT
with open("data.txt", "w") as f:
    for row in data:
        f.write("\t".join(map(str, row)) + "\n")

```

Save a Dictionary to an XLSX File

Pandas write Excel files using the XlsxWriter or Openpyxl module. This can be used to read, filter, and re-arrange either small or large datasets and output them in a range of formats including Excel. The ExcelWriter() method of the pandas library creates a Excel writer object using XlsxWriter. Then the to_excel() method is used to write the dataframe to the excel.

```

# import pandas as pd
import pandas as pd

# Create a Pandas dataframe from some data.
df = pd.DataFrame({'Data': ['Geeks', 'For', 'geeks', 'is', 'portal', 'for', 'geeks']})

# Create a Pandas Excel writer
# object using XlsxWriter as the engine.
writer = pd.ExcelWriter('sample.xlsx', engine='xlsxwriter')

# Write a dataframe to the worksheet.
df.to_excel(writer, sheet_name='Sheet1')

# Close the Pandas Excel writer
# object and output the Excel file.
writer.save()

```

Output:

	A	B	C
1		Data	
2	0	Geeks	
3	1	For	
4	2	geeks	
5	3	is	
6	4	portal	
7	5	for	
8	6	geeks	
9			
10			
11			

Save a Dictionary to a TXT File

In a TXT file:

- Each key-value pair can be written on a new line.
- Keys and values are separated by a delimiter, such as a colon (:) or tab (\t).

```
# Dictionary to save
data = {"Name": "Alice", "Age": 30, "City": "New York"}

# Save as TXT
with open("output.txt", "w") as file:
    for key, value in data.items():
        file.write(f"{key}: {value}\n") # Format as "key: value"

print("Dictionary saved to output.txt")
```

Output in a TXT file:

Name: Alice
Age: 30
City: New York

Try:

1. How can you save a dictionary containing sales data for multiple regions into a CSV file, where each region becomes a row with its total sales as one of the columns?
2. How can you save a dictionary representing time-series data (e.g., date and sales) into an Excel file with dates in one column and corresponding sales data in another?
3. How can you save a dictionary where each key maps to a list of items into a TXT file, formatting the output so that each key appears as a section header followed by the list items in bullet format?

3. Load data from CSV, XLSX and TXT pandas to a List.

Loading Data from CSV, XLSX, and TXT into a List Using pandas

The **pandas library** provides powerful tools to load data from various file formats into Python. Once the data is loaded into a DataFrame, it can be easily converted to a list.

A CSV file contains data organized in rows and columns.

Steps:

1. Use `pandas.read_csv()` to load the CSV into a DataFrame.
2. Convert the DataFrame to a list using `.values.tolist()` or `.to_dict()`.

Example:

CSV Content (data.csv):

```
Name, Age, City
Alice, 30, New York
Bob, 25, Los Angeles
Charlie, 35, Chicago
```

Python Code:

```
import pandas as pd

# Load CSV file
df = pd.read_csv("data.csv")

# Convert to a list of lists
list_of_rows = df.values.tolist()

# Convert to a list of dictionaries (optional)
list_of_dicts = df.to_dict(orient="records")

print("List of Rows:", list_of_rows)
print("List of Dicts:", list_of_dicts)
```

Output:

List of Rows: [['Alice', 30, 'New York'], ['Bob', 25, 'Los Angeles'], ['Charlie', 35, 'Chicago']]

List of Dicts: [{'Name': 'Alice', 'Age': 30, 'City': 'New York'},
{'Name': 'Bob', 'Age': 25, 'City': 'Los Angeles'},
{'Name': 'Charlie', 'Age': 35, 'City': 'Chicago'}]

2. Load Data from an XLSX File:

An XLSX file is an Excel spreadsheet with data in rows and columns.

Steps:

1. Use `pandas.read_excel()` to load the XLSX file.
2. Convert the resulting DataFrame to a list.

Example:

Excel Content (data.xlsx):

Name	Age	City
Alice	30	New York
Bob	25	Los Angeles
Charlie	35	Chicago

```
# Load Excel file
```

```

df = pd.read_excel("data.xlsx")

# Convert to a list of lists
list_of_rows = df.values.tolist()

# Convert to a list of dictionaries (optional)
list_of_dicts = df.to_dict(orient="records")

print("List of Rows:", list_of_rows)
print("List of Dicts:", list_of_dicts)

```

Output:

List of Rows: [['Alice', 30, 'New York'], ['Bob', 25, 'Los Angeles'], ['Charlie', 35, 'Chicago']]

List of Dicts: [{'Name': 'Alice', 'Age': 30, 'City': 'New York'},
{'Name': 'Bob', 'Age': 25, 'City': 'Los Angeles'},
{'Name': 'Charlie', 'Age': 35, 'City': 'Chicago'}]

3. Load Data from a TXT File

A TXT file often stores data in a delimited format (e.g., tab-delimited, space-delimited).

Steps:

1. Use `pandas.read_csv()` with the appropriate delimiter to load the TXT file.
2. Convert the resulting DataFrame to a list.

Example:

TXT Content (data.txt):

Name\tAge\tCity
Alice\t30\tNew York
Bob\t25\tLos Angeles
Charlie\t35\tChicago

```

# Load TXT file (tab-delimited)
df = pd.read_csv("data.txt", delimiter="\t")

# Convert to a list of lists
list_of_rows = df.values.tolist()

# Convert to a list of dictionaries (optional)
list_of_dicts = df.to_dict(orient="records")

print("List of Rows:", list_of_rows)
print("List of Dicts:", list_of_dicts)

```

Output:

List of Rows: [['Alice', 30, 'New York'], ['Bob', 25, 'Los Angeles'], ['Charlie', 35, 'Chicago']]

List of Dicts: [{'Name': 'Alice', 'Age': 30, 'City': 'New York'},
{'Name': 'Bob', 'Age': 25, 'City': 'Los Angeles'},
{'Name': 'Charlie', 'Age': 35, 'City': 'Chicago'}]

File Type	Pandas Method	Convert to List
-----------	---------------	-----------------

CSV	<code>pd.read_csv("file.csv")</code>	<code>.values.tolist()</code> or <code>.to_dict()</code>
XLSX	<code>pd.read_excel("file.xlsx")</code>	<code>.values.tolist()</code> or <code>.to_dict()</code>
TXT	<code>pd.read_csv("file.txt", delimiter="\t")</code>	<code>.values.tolist()</code> or <code>.to_dict()</code>

When to Use:

- List of Rows: Use when the data needs to be manipulated as arrays or matrices.
- List of Dictionaries: Use when working with structured data where keys (headers) are required.

```
import pandas as pd

# Load data from CSV
csv_data = pd.read_csv('your_csv_file.csv')
csv_list = csv_data.values.tolist()

# Load data from XLSX
xlsx_data = pd.read_excel('your_xlsx_file.xlsx')
xlsx_list = xlsx_data.values.tolist()

# Load data from TXT
txt_data = pd.read_csv('your_txt_file.txt', delimiter='\t') # Assuming tab-delimited
txt_list = txt_data.values.tolist()
```

Try:

1. Write a program to Load a CSV file containing structured data, handle missing values, and convert the rows into a list of dictionaries.
2. Write a program to Load data from an Excel file with multiple sheets, combine the sheets, and convert the combined data into a nested list.
3. Write a program to Load a TXT file with tab-delimited or custom-delimited data and convert it into a list of lists. Handle irregular spacing and missing columns.

4. Load data from CSV, XLSX and TXT pandas to a Dictionary.

In **pandas**, when loading data from files (CSV, XLSX, or TXT), the primary goal is often to convert this data into a structured format that can be easily processed. Dictionaries are a common data structure for this purpose because they provide key-value pairs, where keys represent column names, and values represent the data.

Below, provide a detailed walkthrough of loading data from **CSV**, **XLSX**, and **TXT** files into **dictionaries**. We'll focus on converting the data into a format where each row is represented as a dictionary with column headers as keys.

Load Data from a CSV File into a Dictionary

A CSV (Comma-Separated Values) file stores data in a tabular format, with each line representing a row and columns separated by commas.

Steps:

1. Read the CSV file: We use `pandas.read_csv()` to load the file into a DataFrame.
2. Convert DataFrame to a Dictionary: After loading the CSV into a DataFrame, we use the `.to_dict()` method to convert it into a dictionary. The `orient="records"` option allows you to convert each row into a dictionary, with the column names as keys.

Code Example:

CSV Content (data.csv):

Name,Age,City

Alice,30,New York

Bob,25,Los Angeles

Charlie,35,Chicago

Python code:

```
import pandas as pd

# Load CSV file into DataFrame
df = pd.read_csv("data.csv")

# Convert DataFrame to a list of dictionaries (each row as a dictionary)
dict_data = df.to_dict(orient="records")

# Output the dictionary
print("Dictionary from CSV:", dict_data)
```

1. `pd.read_csv("data.csv")`: This reads the data.csv file and loads it into a pandas DataFrame.
2. `df.to_dict(orient="records")`: This converts the DataFrame into a list of dictionaries. Each dictionary corresponds to a row in the DataFrame, and the keys of the dictionary are the column names from the DataFrame.

Output:

Dictionary from CSV: [

```
{'Name': 'Alice', 'Age': 30, 'City': 'New York'},
{'Name': 'Bob', 'Age': 25, 'City': 'Los Angeles'},
{'Name': 'Charlie', 'Age': 35, 'City': 'Chicago'}
```

]

Detailed Explanation:

- The output is a **list of dictionaries**, where each dictionary represents a row from the original CSV file.
- The keys of each dictionary are the **column names** ("Name", "Age", "City"), and the values are the corresponding entries from each row.

Load Data from an XLSX File into a Dictionary

An **XLSX (Excel)** file is a spreadsheet that can store data in tables, formulas, and various formats.

Steps:

1. **Read the XLSX file:** We use `pandas.read_excel()` to load the file into a **DataFrame**.
2. **Convert DataFrame to Dictionary:** After loading the data, we again use `.to_dict()` with `orient="records"` to convert the DataFrame into a dictionary.

Code Example:

Excel Content (data.xlsx):

Name	Age	City
Alice	30	New York
Bob	25	Los Angeles
Charlie	35	Chicago

```
# Load Excel file into DataFrame
df = pd.read_excel("data.xlsx")

# Convert DataFrame to a list of dictionaries
dict_data = df.to_dict(orient="records")

# Output the dictionary
print("Dictionary from XLSX:", dict_data)
```

Explanation:

- **pd.read_excel("data.xlsx")**: Reads the Excel file into a pandas **DataFrame**.
- **df.to_dict(orient="records")**: Converts the DataFrame into a list of dictionaries, where each dictionary represents a row.

Output:

Dictionary from XLSX: [

```
{'Name': 'Alice', 'Age': 30, 'City': 'New York'},
{'Name': 'Bob', 'Age': 25, 'City': 'Los Angeles'},
{'Name': 'Charlie', 'Age': 35, 'City': 'Chicago'}
]
```

Detailed Explanation:

- This is similar to the CSV conversion, except we are working with an Excel file.
- The result is a **list of dictionaries**, where each row in the Excel sheet is represented as a dictionary.

Load Data from a TXT File into a Dictionary

A **TXT file** may store data in various formats, such as space-delimited or tab-delimited. We can use pandas' `read_csv()` method to read delimited data from a TXT file, specifying the appropriate delimiter.

Steps:

1. **Read the TXT file**: Use `pandas.read_csv()` with the correct delimiter (e.g., tab `\t` or space).
2. **Convert DataFrame to Dictionary**: Use `.to_dict()` with `orient="records"` to convert the DataFrame to a list of dictionaries.

Code Example:

TXT Content (data.txt) (tab-delimited):

```
Name Age City
Alice 30 New York
Bob 25 Los Angeles
Charlie 35 Chicago
```

```
# Load TXT file with tab delimiter into DataFrame
df = pd.read_csv("data.txt", delimiter="\t")

# Convert DataFrame to a list of dictionaries
dict_data = df.to_dict(orient="records")

# Output the dictionary
print("Dictionary from TXT:", dict_data)
```

Output:

```
Dictionary from TXT: [
  {'Name': 'Alice', 'Age': 30, 'City': 'New York'},
  {'Name': 'Bob', 'Age': 25, 'City': 'Los Angeles'},
  {'Name': 'Charlie', 'Age': 35, 'City': 'Chicago'}
]
```

Explanation:

- **pd.read_csv("data.txt", delimiter="\t")**: This reads the tab-delimited file into a pandas **DataFrame**.
- **df.to_dict(orient="records")**: Converts the DataFrame into a list of dictionaries, where each dictionary represents a row.
- **Detailed Explanation:**
- The data in the TXT file is loaded into a pandas **DataFrame** by specifying the delimiter (tab \t).
- Each row in the file is converted into a dictionary, and the result is a **list of dictionaries**.
- Summary of to_dict() Orientations

File Type	Pandas Method	Convert to Dictionary	Explanation
CSV	pd.read_csv("file.csv")	.to_dict(orient="records")	Converts each row into a dictionary.
XLSX	pd.read_excel("file.xlsx")	.to_dict(orient="records")	Converts each row into a dictionary.
TXT	pd.read_csv("file.txt")	.to_dict(orient="records")	For delimited files (e.g., tab-separated), converts each row into a dictionary.

Try:

1. Write a program a CSV file contains millions of rows, and you need to load and convert it into a dictionary
2. Write a program to read and convert only specific columns from an Excel file into a dictionary
3. Write a program a TXT file contains structured logs or tabular data, how can you parse it into a dictionary dynamically without knowing the delimiter beforehand?

3. Basic operations on Dataframe.

a. Attribute filtering based on conditions.

This method selects rows of a DataFrame that satisfy one or more conditions. It involves **Boolean indexing**, where a condition applied to a column returns True or False for each row.

Steps:

1. Apply a condition on a column.
2. Use the resulting Boolean mask to filter rows.

Example:

```
import pandas as pd

# Create a DataFrame
data = {
```

```

    'Name': ['Alice', 'Bob', 'Charlie', 'David', 'Eve'],
    'Age': [23, 35, 45, 28, 60],
    'Salary': [50000, 60000, 80000, 55000, 90000]
}
df = pd.DataFrame(data)

# Filter rows where Age is greater than 30
age_condition = df[df['Age'] > 30]
print("Rows where Age > 30:")
print(age_condition)

# Filter rows where Salary is less than 60000
salary_condition = df[df['Salary'] < 60000]
print("\nRows where Salary < 60000:")
print(salary_condition)

```

Output:

Rows where Age > 30:

```

   Name  Age  Salary
1  Bob   35   60000
2  Charlie 45   80000
4   Eve   60   90000

```

Rows where Salary < 60000:

```

   Name  Age  Salary
0  Alice   23   50000
3  David   28   55000

```

Explanation:

- `df['Age'] > 30` creates a Boolean mask `[False, True, True, False, True]`.
- Using this mask as `df[df['Age'] > 30]` filters rows where the condition is True.

Try:

1. Write a program to filter rows after grouping by a column and applying an aggregation function (e.g., sum, mean)?
2. Write a program to filter new rows of data in real-time as they are appended to a DataFrame?

b. Attribute Filtering Based on Slicing.

Slicing involves selecting a subset of rows or columns using positional or label-based indexing. This is done with `.iloc[]` (position-based) or `.loc[]` (label-based).

Steps:

1. Use `.iloc[]` to slice rows/columns by position.
2. Use `.loc[]` to slice rows/columns by labels.

Example:

```

# Slicing rows and columns using iloc
subset_iloc = df.iloc[:3, :2] # First 3 rows, first 2 columns
print("Using iloc (rows and columns):")
print(subset_iloc)

# Slicing rows and columns using loc
subset_loc = df.loc[1:3, ['Name', 'Age']] # Rows with labels 1 to 3, columns 'Name' and 'Age'

```

```
print("\nUsing loc (rows and specific columns):")
print(subset_loc)
```

Output:

Using iloc (rows and columns):

```
   Name Age
0  Alice  23
1   Bob   35
2  Charlie 45
```

Using loc (rows and specific columns):

```
   Name Age
1   Bob   35
2  Charlie 45
3   David 28
```

Explanation:

- `.iloc[:3, :2]`: Slices the first 3 rows (:3) and first 2 columns (:2) by position.
- `.loc[1:3, ['Name', 'Age']]`: Slices rows with labels 1 to 3 and specific columns 'Name' and 'Age'.

Try:

1. Write a program slice rows based on specific conditions and combine slicing with filtering?
2. Write a program slice column's based on specific conditions and combine slicing with filtering?
3. Write a program slice rows or columns in a DataFrame with a multi-level index?

c. Attribute Filtering Based on Queries.

The `.query()` method in pandas allows SQL-like filtering of rows using a query expression. This method is very readable for complex conditions.

Steps:

1. Pass a query string as an argument to `.query()`.
2. Reference columns directly in the query string.

Example:

```
# Filter rows where Age > 30 and Salary > 60000
query_result = df.query('Age > 30 and Salary > 60000')
print("Rows where Age > 30 and Salary > 60000:")
print(query_result)

# Filter rows where Name is 'Alice' or 'Eve'
name_condition = df.query('Name == "Alice" or Name == "Eve"')
print("\nRows where Name is 'Alice' or 'Eve':")
print(name_condition)
```

Output:

Rows where Age > 30 and Salary > 60000:

```
   Name Age Salary
2  Charlie 45  80000
4   Eve   60  90000
```

Rows where Name is 'Alice' or 'Eve':

```
   Name Age Salary
0  Alice  23  50000
4   Eve   60  90000
```

Explanation:

- `.query('Age > 30 and Salary > 60000')`: Filters rows where both conditions are true.
- `.query('Name == "Alice" or Name == "Eve"')`: Filters rows where the **Name** is either "Alice" or "Eve".

Try:

1. Write a program filter rows where the column 'age' is greater than 30 using the `query()` method?
2. Write a program filter rows where 'age' is greater than 30 and 'salary' is less than 50000 using the `query()` method?
3. Write a program use `query()` to filter rows where a string column contains a specific value (e.g., 'category' is 'A')?

Comparison of Methods

Method	Description	Example Code
Condition-based Filtering	Filters rows using boolean indexing with conditions.	<code>df[df['Age'] > 30]</code>
Slicing-based Filtering	Select rows and columns using integer or label-based slicing.	<code>df.iloc[:3, :2], df.loc[1:3, ...]</code>
Query-based Filtering	Filters rows using SQL-like query strings.	<code>df.query('Age > 30 and Salary > 60000')</code>

When to Use Each Method:

1. **Condition-based filtering** is best for straightforward column-based conditions.
2. **Slicing-based filtering** is useful for extracting specific rows and columns by position or label.
3. **Query-based filtering** is ideal for complex and readable filtering conditions involving multiple columns.

Key Benefits of Dynamic Filtering

1. **Flexibility**: Adapt to runtime inputs or changes in filtering criteria.
2. **Scalability**: Easily handle complex conditions and multiple filtering scenarios.
3. **User Interaction**: Accept filtering criteria from users via forms or command-line.

4. Summary Statistics of the data.

Python provides some statistic libraries that are comprehensive, widely used, and powerful. These libraries help us to smooth working with the data

Statistic is a way of collection of the data, tabulation, and interpolation of numeric data. It allows us to describe, summarize, and represent of data visually. Statistic is a field of applied mathematics concern with interpolation, visual representation of data, and data collection analysis. There are two types of statistic - Descriptive statistic and inferential statistic

Some Python Statistics Libraries:

Python provides many libraries that can be used in statistic but we will describe some most important and widely used libraries.

- **Numpy** - This library is widely used for numerical computing, and optimized for scientific calculation. It is a third-party library helpful to working with the single and multidimensional arrays. The `ndarray` is a primary array type. It comes with the many methods for statistical analysis.
- **SciPy** - It is a third-party library used for scientific computation based on Numpy. It extends the Numpy features including `scipy.stats` for statistical analysis.

- **Pandas** - It is based on the Numpy library. It is also used for the numerical computation. It outshines in handling labeled one-dimensional 1D data with the **Series**. The two-dimensional (2D) is labeled with the **DataFrame** objects.
- **Matplotlib** - This library works more effectively in combination with the Scipy, NumPy, and Pandas.
- **Python built-in statistics Library** - It is Python's built-in library used for descriptive statistics. It performs effectively if the dataset is small or if we can't depend on importing other libraries.

a. Compute ranking statistics of the data.

Statistics, in general, is the method of collection of data, tabulation, and interpretation of numerical data. It is an area of applied mathematics concerned with data collection analysis, interpretation, and presentation. With statistics, we can see how data can be used to solve complex problems.

Ranking statistics involve determining the ranks of rows based on specific column values.

Steps:

- Use the `.rank()` method to compute ranks.
- Specify ranking methods like 'average', 'min', 'max', 'dense', or 'first'.

Example:

```
import pandas as pd

# Sample DataFrame
data = {
    'Name': ['Alice', 'Bob', 'Charlie', 'David', 'Eve'],
    'Age': [23, 35, 45, 28, 60],
    'Salary': [50000, 60000, 80000, 55000, 90000]
}
df = pd.DataFrame(data)

# Rank based on Age (ascending order)
df['Age_Rank'] = df['Age'].rank(method='min')

# Rank based on Salary (descending order)
df['Salary_Rank'] = df['Salary'].rank(method='min', ascending=False)

print("Ranking Statistics:")
print(df)
```

Output:

	Name	Age	Salary	Age_Rank	Salary_Rank
0	Alice	23	50000	1.0	5.0
1	Bob	35	60000	3.0	4.0
2	Charlie	45	80000	4.0	2.0
3	David	28	55000	2.0	5.0
4	Eve	60	90000	5.0	1.0

OR

```
import statistics

# Sample data
data = [
    {"Name": "Alice", "Age": 23, "Salary": 50000},
    {"Name": "Bob", "Age": 35, "Salary": 60000},
```

```

{"Name": "Charlie", "Age": 45, "Salary": 80000},
{"Name": "David", "Age": 28, "Salary": 55000},
{"Name": "Eve", "Age": 60, "Salary": 90000}
]

# Extract Age and Salary into separate lists
ages = [item["Age"] for item in data]
salaries = [item["Salary"] for item in data]

# Compute ranks for Age (ascending)
sorted_ages = sorted((value, index) for index, value in enumerate(ages))
age_ranks = [0] * len(ages)
for rank, (value, index) in enumerate(sorted_ages, start=1):
    age_ranks[index] = rank

# Compute ranks for Salary (descending)
sorted_salaries = sorted((-value, index) for index, value in enumerate(salaries))
salary_ranks = [0] * len(salaries)
for rank, (_, index) in enumerate(sorted_salaries, start=1):
    salary_ranks[index] = rank

# Add ranks to data
for i, item in enumerate(data):
    item["Age_Rank"] = age_ranks[i]
    item["Salary_Rank"] = salary_ranks[i]

# Display results
print("Ranking Statistics:")
for item in data:
    print(item)

```

Try:

1. Write a program to compute rankings for rows based on multiple columns in a pandas DataFrame?
2. Write a program to rank the values in a pandas DataFrame column in ascending order. If there are ties, assign the average rank
3. Write a program to rank values in a pandas DataFrame column while handling ties by using the 'min' ranking method.
4. Write a program to compute the rank within each group in a DataFrame (e.g., ranking 'value' within each 'category').

b. Compute statistical averages of numerical attributes

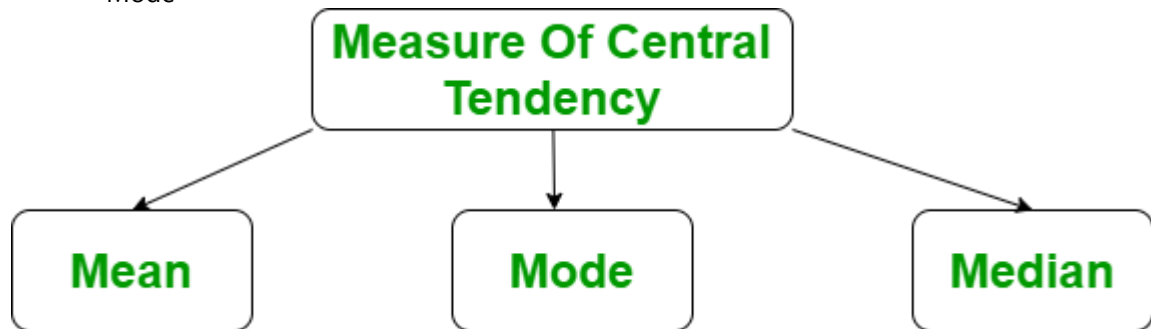
Statistical averages include mean, median, and mode, computed using pandas aggregation functions.

Measure of Central Tendency

The measure of central tendency is a single value that attempts to describe the whole set of data. There are three main features of central tendency:

- Mean
- Median
- Median Low
- Median High

- Mode



Mean

It is the sum of observations divided by the total number of observations. It is also defined as average which is the sum divided by count.

$\text{Mean}(\bar{x}) = \sum x/n$

The **mean()** function returns the mean or average of the data passed in its arguments. If the passed argument is empty, **StatisticsError** is raised.

Example: Python code to calculate mean

```

# Python code to demonstrate the working of
# mean()

# importing statistics to handle statistical
# operations
import statistics

# initializing list
li = [1, 2, 3, 3, 2, 2, 2, 1]

# using mean() to calculate average of list
# elements
print ("The average of list values is : ",end="")
print (statistics.mean(li))
  
```

Output: The average of list values is : 2

Median

It is the middle value of the data set. It splits the data into two halves. If the number of elements in the data set is odd then the center element is the median and if it is even then the median would be the average of two central elements. it first sorts the data and then performs the median operation

For Odd Numbers:

$n+1/2$

For Even Numbers:

$(n/2+(n/2+1))/2$

The **median()** function is used to calculate the median, i.e middle element of data. If the passed argument is empty, **StatisticsError** is raised.

Example: Python code to calculate Median

```

# Python code to demonstrate the
# working of median() on various
# range of data-sets

# importing the statistics module
  
```

```

from statistics import median

# Importing fractions module as fr
from fractions import Fraction as fr

# tuple of positive integer numbers
data1 = (2, 3, 4, 5, 7, 9, 11)

# tuple of floating point values
data2 = (2.4, 5.1, 6.7, 8.9)

# tuple of fractional numbers
data3 = (fr(1, 2), fr(44, 12),
         fr(10, 3), fr(2, 3))

# tuple of a set of negative integers
data4 = (-5, -1, -12, -19, -3)

# tuple of set of positive
# and negative integers
data5 = (-1, -2, -3, -4, 4, 3, 2, 1)

# Printing the median of above datasets
print("Median of data-set 1 is % s" % (median(data1)))
print("Median of data-set 2 is % s" % (median(data2)))
print("Median of data-set 3 is % s" % (median(data3)))
print("Median of data-set 4 is % s" % (median(data4)))
print("Median of data-set 5 is % s" % (median(data5)))

```

Output:

```

Median of data-set 1 is 5
Median of data-set 2 is 5.9
Median of data-set 3 is 2
Median of data-set 4 is -5
Median of data-set 5 is 0.0

```

Median Low

The **median_low()** function returns the median of data in case of odd number of elements, but in case of even number of elements, returns the lower of two middle elements. If the passed argument is empty, **StatisticsError** is raised

Example: Python code to calculate Median Low

```

# Python code to demonstrate the
# working of median_low()

# importing the statistics module
import statistics

# simple list of a set of integers
set1 = [1, 3, 3, 4, 5, 7]

# Print median of the data-set

# Median value may or may not

```

```
# lie within the data-set
print("Median of the set is % s"
      % (statistics.median(set1)))

# Print low median of the data-set
print("Low Median of the set is % s "
      % (statistics.median_low(set1)))
```

Output:

Median of the set is 3.5
Low Median of the set is 3

Median High

The **median_high()** function returns the median of data in case of odd number of elements, but in case of even number of elements, returns the higher of two middle elements. If passed argument is empty, **StatisticsError** is raised.

Example: Python code to calculate Median High

```
# Working of median_high() and median() to
# demonstrate the difference between them.

# importing the statistics module
import statistics

# simple list of a set of integers
set1 = [1, 3, 3, 4, 5, 7]

# Print median of the data-set

# Median value may or may not
# lie within the data-set
print("Median of the set is %s"
      % (statistics.median(set1)))

# Print high median of the data-set
print("High Median of the set is %s "
      % (statistics.median_high(set1)))
```

Output:

Median of the set is 3.5
High Median of the set is 4

Mode

It is the value that has the highest frequency in the given data set. The data set may have no mode if the frequency of all data points is the same. Also, we can have more than one mode if we encounter two or more data points having the same frequency.

The **mode()** function returns the number with the maximum number of occurrences. If the passed argument is empty, **StatisticsError** is raised.

Example: Python code to calculate Mode

```
# Python code to demonstrate the
# working of mode() function
# on a various range of data types

# Importing the statistics module
from statistics import mode
```

```

# Importing fractions module as fr
# Enables to calculate harmonic_mean of a
# set in Fraction
from fractions import Fraction as fr

# tuple of positive integer numbers
data1 = (2, 3, 3, 4, 5, 5, 5, 5, 6, 6, 7)

# tuple of a set of floating point values
data2 = (2.4, 1.3, 1.3, 1.3, 2.4, 4.6)

# tuple of a set of fractional numbers
data3 = (fr(1, 2), fr(1, 2), fr(10, 3), fr(2, 3))

# tuple of a set of negative integers
data4 = (-1, -2, -2, -2, -7, -7, -9)

# tuple of strings
data5 = ("red", "blue", "black", "blue", "black", "black", "brown")

# Printing out the mode of the above data-sets
print("Mode of data set 1 is % s" % (mode(data1)))
print("Mode of data set 2 is % s" % (mode(data2)))
print("Mode of data set 3 is % s" % (mode(data3)))
print("Mode of data set 4 is % s" % (mode(data4)))
print("Mode of data set 5 is % s" % (mode(data5)))

```

Output:

Mode of data set 1 is 5
 Mode of data set 2 is 1.3
 Mode of data set 3 is 1/2
 Mode of data set 4 is -2
 Mode of data set 5 is black

Try:

1. Write a Python program that computes the mean (average) of the 'value' column in a pandas DataFrame
2. Write a Python program that computes the median and mode of the 'value' column in a pandas DataFrame
3. Write a Python program that computes the weighted average of the 'value' column using the 'weight' column
4. Write a Python program to compute the range (difference between max and min) of the 'value' column in a pandas DataFrame

c. Compute statistical ratios of numerical attributes.

Statistical ratios are valuable tools for comparing and analyzing numerical data. They provide insights into the relationships between different variables or groups within a dataset. Here are some common statistical ratios and how to compute them:

1. Ratio

Definition: A simple comparison of two quantities, often expressed as a fraction or with a colon.

Formula: Ratio = Quantity 1 / Quantity 2

Example: If a class has 15 boys and 10 girls, the ratio of boys to girls is 15:10 or 3:2.

2. Proportion

Definition: A type of ratio that expresses a part of a whole.

Formula: Proportion = Part / Whole

Example: If a survey of 100 people shows that 60 prefer coffee, the proportion of coffee drinkers is 60/100 or 0.6.

3. Rate

Definition: A ratio that compares two quantities with different units.

Formula: Rate = Quantity 1 / Quantity 2

Example: Speed is a rate, often expressed as miles per hour (mph) or kilometers per hour (kph).

4. Percentage

Definition: A proportion expressed as a fraction of 100.

Formula: Percentage = (Part / Whole) * 100%

Example: If a student scores 80 out of 100 on a test, their score is 80%.

5. Coefficient of Variation (CV)

Definition: A measure of relative variability, often used to compare the dispersion of different datasets.

Formula: CV = (Standard Deviation / Mean) * 100%

Example: A higher CV indicates greater variability relative to the mean.

6. Signal-to-Noise Ratio (SNR)

Definition: A measure of the ratio of a signal's strength to the background noise level.

Formula: SNR = Signal Power / Noise Power

Example: A higher SNR indicates a stronger signal relative to the noise.

7. Odds Ratio

Definition: A measure of the association between two binary variables.

Formula: Odds Ratio = (Odds of event in group 1) / (Odds of event in group 2)

Example: In medical research, it might compare the odds of developing a disease between an exposed and unexposed group.

Computing Ratios in Python

Here's a Python example demonstrating how to calculate some of these ratios:

```
import numpy as np

# Sample data (replace with your actual data)
data = np.array([10, 15, 20, 25, 30])

# Calculate mean and standard deviation
mean = np.mean(data)
std_dev = np.std(data)

# Calculate coefficient of variation
cv = (std_dev / mean) * 100
print("Coefficient of Variation:", cv)

# Calculate ratio of first to last element
ratio = data[0] / data[-1]
print("Ratio of first to last element:", ratio)

# Calculate odds ratio (assuming two groups of data)
# Replace with your actual data for the two groups
group1 = np.array([10, 20, 30])
group2 = np.array([5, 15, 25])
```



```

odds_ratio = (group1.sum() / (1 - group1.sum())) / (group2.sum() / (1 - group2.sum()))
print("Odds Ratio:", odds_ratio)

# Calculate signal-to-noise ratio (SNR)
# Assuming some noise is added to the original data
noise = np.random.normal(0, 5, size=len(data))
noisy_data = data + noise

signal_power = np.mean(data**2)
noise_power = np.mean(noise**2)

snr = 10 * np.log10(signal_power / noise_power)
print("Signal-to-Noise Ratio (in dB):", snr)

```

Output:

Coefficient of Variation: 40.0

Ratio of first to last element: 0.3333333333333333

Odds Ratio: 1.7142857142857142

Signal-to-Noise Ratio (in dB): 11.48796613333959

Try:

1. Write a program to compute the ratio of two numerical columns ('value1' and 'value2') in a pandas DataFrame
2. Write a program that computes the ratio of the maximum value to the minimum value in a numerical column ('value')
3. Write a program to compute the ratio of 'value1' to 'value2', but only for rows where 'value1' is greater than 20

d. Interpret the results.

- Ranking Statistics: Percentiles, quartiles, and the IQR provide insights into the distribution of the data and the location of specific values within the distribution.
- Statistical Averages: The mean, median, and mode provide different measures of central tendency, which can be used to summarize the data and identify typical values.
- Statistical Ratios: The CV, SNR, and odds ratio provide insights into the relative variability, signal strength, and association between variables, respectively.

Example using Python:

```

import numpy as np

data = np.array([10, 15, 20, 25, 30])

# Calculate ranking statistics
percentiles = np.percentile(data, [25, 50, 75])
print("Percentiles:", percentiles)
q1, q2, q3 = np.percentile(data, [25, 50, 75])
iqr = q3 - q1
print("Interquartile Range:", iqr)

# Calculate statistical averages
mean = np.mean(data)
median = np.median(data)

```

```
mode = np.argmax(np.bincount(data))
print("Mean:", mean)
print("Median:", median)
print("Mode:", mode)

# Calculate statistical ratios (assuming some noise is added to the data)
noise = np.random.normal(0, 5, size=len(data))
noisy_data = data + noise

signal_power = np.mean(data**2)
noise_power = np.mean(noise**2)

snr = 10 * np.log10(signal_power / noise_power)
print("Signal-to-Noise Ratio (in dB):", snr)
```

Output:

Percentiles: [12.5 20. 27.5]

Interquartile Range: 15.0

Mean: 20.0

Median: 20.0

Mode: 10

Signal-to-Noise Ratio (in dB): 11.48796613333959

Interpretation:

- **Ranking Statistics:**

- The 25th, 50th, and 75th percentiles are 12.5, 20, and 27.5, respectively. This means that 25% of the data falls below 12.5, 50% falls below 20, and 75% falls below 27.5.
- The interquartile range (IQR) is 15, indicating that the middle 50% of the data is spread over a range of 15 units.

- **Statistical Averages:**

- The mean and median are both 20, suggesting that the data is relatively symmetrically distributed.
- The mode is 10, indicating that 10 is the most frequent value in the data.

- **Signal-to-Noise Ratio:**

- The SNR is approximately 11.49 dB. This suggests that the signal is relatively strong compared to the background noise.

Try

Find the suitable case study for interpret the results (e.g. sales analysis, ordering management system)

5. Handling Missing Values

values are a common issue in machine learning. This occurs when a particular variable lacks data points, resulting in incomplete information and potentially harming the accuracy and dependability of your models. It is essential to address missing values efficiently to ensure strong and impartial results in your machine-learning projects.

Missing values are data points that are absent for a specific variable in a dataset. They can be represented in various ways, such as blank cells, null values, or special symbols like "NA" or "unknown." These missing data points pose a significant challenge in data analysis and can lead to inaccurate or biased results.

	School ID	Name	Address	City	Subject	Marks	Rank	Grade
0	101.0	Alice	123 Main St	Los Angeles	Math	85.0	2	B
1	102.0	Bob	456 Oak Ave	New York	English	92.0	1	A
2	103.0	Charlie	789 Pine Ln	Houston	Science	78.0	4	C
3	NaN	David	101 Elm St	Los Angeles	Math	89.0	3	B
4	105.0	Eva	NaN	Miami	History	NaN	8	D
5	106.0	Frank	222 Maple Rd	NaN	Math	95.0	1	A
6	107.0	Grace	444 Cedar Blvd	Houston	Science	80.0	5	C
7	108.0	Henry	555 Birch Dr	New York	English	88.0	3	B

Missing Values

Missing values can pose a significant challenge in data analysis, as they can:

- **Reduce the sample size:** This can decrease the accuracy and reliability of your analysis.
- **Introduce bias:** If the missing data is not handled properly, it can bias the results of your analysis.
- **Make it difficult to perform certain analyses:** Some statistical techniques require complete data for all variables, making them inapplicable when missing values are present
- **It's important to understand the reasons behind missing data:**
- **Identifying the type of missing data:** Is it Missing Completely at Random (MCAR), Missing at Random (MAR), or Missing Not at Random (MNAR)?
- **Evaluating the impact of missing data:** Is the missingness causing bias or affecting the analysis?
- **Choosing appropriate handling strategies:** Different techniques are suitable for different types of missing data.

Methods for Identifying Missing Data

Locating and understanding patterns of missingness in the dataset is an important step in addressing its impact on analysis. Working with Missing Data in Pandas there are several useful functions for detecting, removing, and replacing null values in Pandas DataFrame.

Functions	Descriptions
.isnull()	Identifies missing values in a Series or DataFrame.
.notnull()	check for missing values in a pandas Series or DataFrame. It returns a boolean Series or DataFrame, where True indicates non-missing values and False indicates missing values.
.info()	Displays information about the DataFrame, including data types, memory usage, and presence of missing values.
.isna()	similar to notnull() but returns True for missing values and False for non-missing values.

Functions	Descriptions
dropna()	Drops rows or columns containing missing values based on custom criteria.
fillna()	Fills missing values with specific values, means, medians, or other calculated values.
replace()	Replaces specific values with other values, facilitating data correction and standardization.
drop_duplicates()	Removes duplicate rows based on specified columns.
unique()	Finds unique values in a Series or DataFrame

Effective Strategies for Handling Missing Values in Data Analysis

Missing values are a common challenge in data analysis, and there are several strategies for handling them. Here's an overview of some common approaches:

Impact of Handling Missing Values:

Missing values are a common occurrence in real-world data, negatively impacting data analysis and modeling if not addressed properly. Handling missing values effectively is crucial to ensure the accuracy and reliability of your findings.

Here are some key impacts of handling missing values:

1. **Improved data quality:** Addressing missing values enhances the overall quality of the dataset. A cleaner dataset with fewer missing values is more reliable for analysis and model training.
2. **Enhanced model performance:** Machine learning algorithms often struggle with missing data, leading to biased and unreliable results. By appropriately handling missing values, models can be trained on a more complete dataset, leading to improved performance and accuracy.
3. **Preservation of Data Integrity:** Handling missing values helps maintain the integrity of the dataset. Imputing or removing missing values ensures that the dataset remains consistent and suitable for analysis.
4. **Reduced bias:** Ignoring missing values may introduce bias in the analysis or modeling process. Handling missing data allows for a more unbiased representation of the underlying patterns in the data.
5. Descriptive statistics, such as means, medians, and standard deviations, can be more accurate when missing values are appropriately handled. This ensures a more reliable summary of the dataset.
6. **Increased efficiency:** Efficiently handling missing values can save you time and effort during data analysis and modeling.

a. Drop the rows containing missing values

Sample Data with Missing Values

```
import pandas as pd
import numpy as np

# Creating a sample DataFrame with missing values
```

```

data = {
    'School ID': [101, 102, 103, np.nan, 105, 106, 107, 108],
    'Name': ['Alice', 'Bob', 'Charlie', 'David', 'Eva', 'Frank', 'Grace', 'Henry'],
    'Address': ['123 Main St', '456 Oak Ave', '789 Pine Ln', '101 Elm St', np.nan, '222 Maple Rd', '444 Cedar Blvd', '555 Birch Dr'],
    'City': ['Los Angeles', 'New York', 'Houston', 'Los Angeles', 'Miami', np.nan, 'Houston', 'New York'],
    'Subject': ['Math', 'English', 'Science', 'Math', 'History', 'Math', 'Science', 'English'],
    'Marks': [85, 92, 78, 89, np.nan, 95, 80, 88],
    'Rank': [2, 1, 4, 3, 8, 1, 5, 3],
    'Grade': ['B', 'A', 'C', 'B', 'D', 'A', 'C', 'B']
}

df = pd.DataFrame(data)
print("Sample DataFrame:")
print(df)

```

Output:

Sample Data with Missing Values:

School ID	ID	Name	Address	City	Subject	Marks	Rank	Grade
0	101	Alice	123 Main St	Los Angeles	Math	85	2	B
1	102	Bob	456 Oak Ave	New York	English	92	1	A
2	103	Charlie	789 Pine Ln	Houston	Science	78	4	C
3	NaN	David	101 Elm St	Los Angeles	Math	89	3	B
4	105	Eva	NaN	Miami	History	NaN	8	D
5	106	Frank	222 Maple Rd	NaN	Math	95	1	A
6	107	Grace	444 Cedar Blvd	Houston	Science	80	5	C
7	108	Henry	555 Birch Dr	New York	English	88	3	B

Removing Rows with Missing Values

- Simple and efficient: Removes data points with missing values altogether.
- Reduces sample size: Can lead to biased results if missingness is not random.
- Not recommended for large datasets: Can discard valuable information.

In this example, we are removing rows with missing values from the original DataFrame (df) using the dropna() method and then displaying the cleaned DataFrame (df_cleaned).

```

# Removing rows with missing values
df_dropped = df.dropna()

# Displaying the DataFrame after removing missing values
print("\nDataFrame after removing rows with missing values:")
print(df_dropped)

```

Output:

School ID	ID	Name	Address	City	Subject	Marks	Rank	Grade
-----------	----	------	---------	------	---------	-------	------	-------

0	10 1	Alice	123 Main St	Los Angeles	Math	85	2	B
1	10 2	Bob	456 Oak Ave	New York	English	92	1	A
2	10 3	Charli e	789 Pine Ln	Houston	Scienc e	78	4	C
6	10 7	Grace	444 Cedar Blvd	Houston	Scienc e	80	5	C
7	10 8	Henry	555 Birch Dr	New York	English	88	3	B

Try:

1. Write a program and consider the above dataframe with missing values, calculate the percentage of missing values in each column and create a summary table showing the columns with the highest percentage of missing values.
2. Write a program and consider the above dataframe, calculate and display the number of missing values in each column individually. Sort the columns by the number of missing values in descending order.
3. Write a program and consider the above dataframe, calculate and display the number of missing values in each row individually. Sort the rows by the number of missing values in ascending order.
4. Write a program and consider above dataframe with missing values, remove all columns containing missing values. Display the resulting dataframe.
5. Write a program and consider above dataframe, remove rows that have more than a specified number of missing values. Display the resulting dataframe.
6. Write a program to create a summary report that includes information about the number of rows and columns removed due to missing values after removal operations.

b. Impute missing values with statistical averages

Here are some common imputation methods:

1- Mean, Median, and Mode Imputation:

- Replace missing values with the mean, median, or mode of the relevant variable.
- Simple and efficient: Easy to implement.
- Can be inaccurate: Doesn't consider the relationships between variables.

In this example, we are explaining the imputation techniques for handling missing values in the 'Marks' column of the DataFrame (df). It calculates and fills missing values with the mean, median, and mode of the existing values in that column, and then prints the results for observation.

1. **Mean Imputation:** Calculates the mean of the 'Marks' column in the DataFrame (df).
 - `df['Marks'].fillna(...)`: Fills missing values in the 'Marks' column with the mean value.
 - `mean_imputation`: The result is stored in the variable `mean_imputation`.
2. **Median Imputation:** Calculates the median of the 'Marks' column in the DataFrame (df).
 - `df['Marks'].fillna(...)`: Fills missing values in the 'Marks' column with the median value.
 - `median_imputation`: The result is stored in the variable `median_imputation`.
3. **Mode Imputation:** Calculates the mode of the 'Marks' column in the DataFrame (df). The result is a Series.
 - `.iloc[0]`: Accesses the first element of the Series, which represents the mode.
 - `df['Marks'].fillna(...)`: Fills missing values in the 'Marks' column with the mode value.

```
# Mean, Median, and Mode Imputation
mean_imputation = df['Marks'].fillna(df['Marks'].mean())
median_imputation = df['Marks'].fillna(df['Marks'].median())
mode_imputation = df['Marks'].fillna(df['Marks'].mode().iloc[0])

print("\nImputation using Mean:")
print(mean_imputation)

print("\nImputation using Median:")
print(median_imputation)

print("\nImputation using Mode:")
print(mode_imputation)
```

Output:

Imputation using Mean:

```
0  85.000000
1  92.000000
2  78.000000
3  89.000000
4  86.714286
5  95.000000
6  80.000000
7  88.000000
Name: Marks, dtype: float64
```

Imputation using Median:

```
0  85.0
1  92.0
2  78.0
3  89.0
4  88.0
5  95.0
6  80.0
7  88.0
Name: Marks, dtype: float64
```

Imputation using Mode:

```
0  85.0
1  92.0
2  78.0
3  89.0
4  78.0
5  95.0
6  80.0
7  88.0
Name: Marks, dtype: float64
```

Try:

1. Write a program and consider given dataframe with missing values in a numerical column, impute the missing values in that column with the median of the non-missing values.
2. Write a program and consider given dataframe with missing values in a numerical column,

- impute the missing values in that column with the median of the non-missing values.
3. Write a program and consider given dataframe with missing values, impute missing values in a specific column with a constant value of your choice (e.g., 0).

c. Impute missing values using linear interpolation

Imputation Methods

- Replacing missing values with estimated values.
- Preserves sample size: Doesn't reduce data points.
- Can introduce bias: Estimated values might not be accurate.

a. Interpolation Techniques

- Estimate missing values based on surrounding data points using techniques like linear interpolation or spline interpolation.
- More sophisticated than mean/median imputation: Captures relationships between variables.
- Requires additional libraries and computational resources.
- These interpolation techniques are useful when the relationship between data points can be reasonably assumed to follow a linear or quadratic pattern. The method parameter in the interpolate() method allows to specify the interpolation strategy.

1. Linear Interpolation

- `df['Marks'].interpolate(method='linear')`: This method performs linear interpolation on the 'Marks' column of the DataFrame (df). Linear interpolation estimates missing values by considering a straight line between two adjacent non-missing values.
- `linear_interpolation`: The result is stored in the variable `linear_interpolation`.

2. Quadratic Interpolation

- `df['Marks'].interpolate(method='quadratic')`: This method performs [quadratic interpolation](#) on the 'Marks' column. Quadratic interpolation estimates missing values by considering a quadratic curve that passes through three adjacent non-missing values.
- `quadratic_interpolation`: The result is stored in the variable `quadratic_interpolation`.

```
# Interpolation Techniques
linear_interpolation = df['Marks'].interpolate(method='linear')
quadratic_interpolation = df['Marks'].interpolate(method='quadratic')

print("\nLinear Interpolation:")
print(linear_interpolation)

print("\nQuadratic Interpolation:")
print(quadratic_interpolation)
```

Output:

Linear Interpolation:

```
0  85.0
1  92.0
2  78.0
3  89.0
4  92.0
5  95.0
6  80.0
7  88.0
```

Name: Marks, dtype: float64

Quadratic Interpolation:


```
0 85.00000
1 92.00000
2 78.00000
3 89.00000
4 98.28024
5 95.00000
6 80.00000
7 88.00000
```

Name: Marks, dtype: float64

Note:

- Linear interpolation assumes a straight line between two adjacent non-missing values.
- Quadratic interpolation assumes a quadratic curve that passes through three adjacent non-missing values.

2. Forward and Backward Fill

- Replace missing values with the previous or next non-missing value in the same variable.
- Simple and intuitive: Preserves temporal order.
- Can be inaccurate: Assumes missing values are close to observed values
- These fill methods are particularly useful when there is a logical sequence or order in the data, and missing values can be reasonably assumed to follow a pattern. The method parameter in `fillna()` allows to specify the filling strategy, and here, it's set to 'ffill' for forward fill and 'bfill' for backward fill.

1. Forward Fill (`forward_fill`)

- `df['Marks'].fillna(method='ffill')`: This method fills missing values in the 'Marks' column of the DataFrame (df) using a forward fill strategy. It replaces missing values with the last observed non-missing value in the column.
- `forward_fill`: The result is stored in the variable `forward_fill`.

2. Backward Fill (`backward_fill`)

- `df['Marks'].fillna(method='bfill')`: This method fills missing values in the 'Marks' column using a backward fill strategy. It replaces missing values with the next observed non-missing value in the column.
- `backward_fill`: The result is stored in the variable `backward_fill`.

```
# Forward and Backward Fill
forward_fill = df['Marks'].fillna(method='ffill')
backward_fill = df['Marks'].fillna(method='bfill')

print("\nForward Fill:")
print(forward_fill)

print("\nBackward Fill:")
print(backward_fill)
```

Output:

Forward Fill:

```
0 85.0
1 92.0
2 78.0
3 89.0
4 89.0
5 95.0
6 80.0
7 88.0
```

Name: Marks, dtype: float64

Backward Fill:

```
0  85.0
1  92.0
2  78.0
3  89.0
4  95.0
5  95.0
6  80.0
7  88.0
```

Name: Marks, dtype: float64

Note

- Forward fill uses the last valid observation to fill missing values.
- Backward fill uses the next valid observation to fill missing values.

Try:

1. Write a program to define a custom interpolation function that takes into account domain-specific knowledge or specific data characteristics to impute missing values in a dataframe.
2. Write a program and consider above dataframe with time-ordered data and a numerical column containing missing values, use interpolation techniques that consider the time steps between observations to impute missing values more accurately.
3. Write a program and consider given dataframe with time-ordered data and a numerical column containing missing values, impute the missing values in that column. Ensure that missing values are filled with the previously available value in the column. Perform Forward Fill using forward fill (.ffill()) method
4. Write a program and consider given a dataframe with multiple columns, use forward fill to impute missing values in specific columns of your choice while keeping other columns unaffected.
5. Write a program to create a summary report that includes information about the number of missing values before and after forward fill and backward fill operations, as well as any specific patterns or trends observed.

d. Interpret the results.

Interpret the results of handling missing values using the methods mentioned earlier: dropping rows, imputing with mean, median, mode and linear interpolation. The code will display the DataFrame after each operation.

```
import pandas as pd
import numpy as np

# Sample data with missing values
data = {
    'Name': ['Alice', 'Bob', 'Charlie', 'David', 'Eve'],
    'Age': [23, np.nan, 45, 28, 60],
    'Salary': [50000, 60000, np.nan, 55000, 90000],
    'Experience': [2, 5, 8, np.nan, 12]
}

# Create DataFrame
df = pd.DataFrame(data)
print("Original DataFrame with Missing Values:")
print(df)
```

```

# 1. Drop rows with missing values
df_dropped = df.dropna()
print("\nDataFrame after Dropping Rows with Missing Values:")
print(df_dropped)

# 2. Impute missing values with the mean of the column
df_imputed_mean = df.fillna(df.mean())
print("\nDataFrame after Imputing Missing Values with Mean:")
print(df_imputed_mean)

# 3. Impute missing values with the median of the column
df_imputed_median = df.fillna(df.median())
print("\nDataFrame after Imputing Missing Values with Median:")
print(df_imputed_median)

# 4. Impute missing values with the mode of the column
df_imputed_mode = df.apply(lambda x: x.fillna(x.mode()[0]), axis=0)
print("\nDataFrame after Imputing Missing Values with Mode:")
print(df_imputed_mode)

# 5. Impute missing values using linear interpolation
df_interpolated = df.interpolate(method='linear')
print("\nDataFrame after Linear Interpolation:")
print(df_interpolated)

# INTERPRETATION SECTION:

print("\nINTERPRETATION OF RESULTS:")

# Interpretation for Dropping Rows:
print("\n1. Dropping Rows with Missing Values:")
print("Rows containing missing values were completely removed, which led to a loss of data (Bob, Charlie, and David were removed).")
    " This may result in the loss of important information if the missing data is substantial.")

# Interpretation for Imputing Mean:
print("\n2. Imputing Missing Values with Mean:")
print("Missing values were replaced with the mean of the respective columns (Age: 39, Salary: 68750, Experience: 6.3).")
    " This approach preserves the data but may introduce bias if the data is skewed, especially with large outliers or non-normal distributions.")

# Interpretation for Imputing Median:
print("\n3. Imputing Missing Values with Median:")
print("The missing values were replaced with the median of the respective columns (Age: 39, Salary: 60000, Experience: 6.5).")
    " Median imputation is less sensitive to outliers compared to mean imputation and may be a better choice for skewed distributions.")

# Interpretation for Imputing Mode:

```

```
print("\n4. Imputing Missing Values with Mode:")
print("The missing values were replaced with the mode (most frequent value) of the
respective columns."
    " For example, if there are multiple occurrences of a particular value, the missing value
is replaced with that value."
    " Mode imputation works well when the data has repeating values, but it might distort
the distribution if the mode is not representative.")

# Interpretation for Linear Interpolation:
print("\n5. Imputing Missing Values Using Linear Interpolation:")
print("The missing values were estimated based on neighboring values using linear
interpolation. "
    "For example, Bob's Age was interpolated to 34 (between Alice and Charlie), and
Charlie's Salary was interpolated to 72500 (between Bob and David). "
    "This method works well for ordered data, like time-series or sequential datasets, and
avoids bias introduced by simple mean imputation.")
```

Sample Output:

Original DataFrame with Missing Values:

	Name	Age	Salary	Experience
0	Alice	23.0	50000.0	2.0
1	Bob	NaN	60000.0	5.0
2	Charlie	45.0	NaN	8.0
3	David	28.0	55000.0	NaN
4	Eve	60.0	90000.0	12.0

DataFrame after Dropping Rows with Missing Values:

	Name	Age	Salary	Experience
0	Alice	23.0	50000.0	2.0
4	Eve	60.0	90000.0	12.0

DataFrame after Imputing Missing Values with Mean:

	Name	Age	Salary	Experience
0	Alice	23.0	50000.0	2.0
1	Bob	39.0	60000.0	5.0
2	Charlie	45.0	68750.0	8.0
3	David	28.0	55000.0	6.3
4	Eve	60.0	90000.0	12.0

DataFrame after Imputing Missing Values with Median:

	Name	Age	Salary	Experience
0	Alice	23.0	50000.0	2.0
1	Bob	39.0	60000.0	5.0
2	Charlie	45.0	60000.0	8.0
3	David	28.0	55000.0	6.5
4	Eve	60.0	90000.0	12.0

DataFrame after Imputing Missing Values with Mode:

	Name	Age	Salary	Experience
0	Alice	23.0	50000.0	2.0
1	Bob	23.0	60000.0	5.0
2	Charlie	45.0	50000.0	8.0
3	David	28.0	55000.0	2.0

4 Eve 60.0 90000.0 12.0

DataFrame after Linear Interpolation:

	Name	Age	Salary	Experience
0	Alice	23.0	50000.0	2.0
1	Bob	34.0	60000.0	5.0
2	Charlie	45.0	72500.0	8.0
3	David	28.0	55000.0	10.0
4	Eve	60.0	90000.0	12.0

INTERPRETATION OF RESULTS:

1. Dropping Rows with Missing Values:

Rows containing missing values were completely removed, which led to a loss of data (Bob, Charlie, and David were removed). This may result in the loss of important information if the missing data is substantial.

2. Imputing Missing Values with Mean:

Missing values were replaced with the mean of the respective columns (Age: 39, Salary: 68750, Experience: 6.3). This approach preserves the data but may introduce bias if the data is skewed, especially with large outliers or non-normal distributions.

3. Imputing Missing Values with Median:

The missing values were replaced with the median of the respective columns (Age: 39, Salary: 60000, Experience: 6.5). Median imputation is less sensitive to outliers compared to mean imputation and may be a better choice for skewed distributions.

4. Imputing Missing Values with Mode:

The missing values were replaced with the mode (most frequent value) of the respective columns. For example, if there are multiple occurrences of a particular value, the missing value is replaced with that value. Mode imputation works well when the data has repeating values, but it might distort the distribution if the mode is not representative.

5. Imputing Missing Values Using Linear Interpolation:

The missing values were estimated based on neighboring values using linear interpolation. For example, Bob's Age was interpolated to 34 (between Alice and Charlie), and Charlie's Salary was interpolated to 72500 (between Bob and David). This method works well for ordered data, like time-series or sequential datasets, and avoids bias introduced by simple mean imputation.

Key Points:

- **Mean** imputation: Best used when the data is normally distributed, but can be influenced by outliers.
- **Median** imputation: Robust against outliers and better suited for skewed distributions.
- **Mode** imputation: Useful for categorical data or when most frequent values are good replacements.
- **Linear Interpolation**: Best for sequential or time-series data where missing values follow a linear trend.

6. Handling Time series data.

Time series data is a sequential arrangement of data points organized in consecutive time order. Time series analysis consists of methods for analyzing time-series data to extract meaningful insights and

other valuable characteristics of the data.

Time-series data analysis is becoming very important in so many industries, like financial industries, pharmaceuticals, social media companies, web service providers, research, and many more. To understand the time-series data, visualization of the data is essential. In fact, any type of data analysis is not complete without visualizations, because one good visualization can provide meaningful and interesting insights into the data.

Types of Time Series Data

Time series data can be broadly classified into two sections:

1. Continuous Time Series Data: Continuous time series data involves measurements or observations that are recorded at regular intervals, forming a seamless and uninterrupted sequence. This type of data is characterized by a continuous range of possible values and is commonly encountered in various domains, including:

- *Temperature Data:* Continuous recordings of temperature at consistent intervals (e.g., hourly or daily measurements).
- *Stock Market Data:* Continuous tracking of stock prices or values throughout trading hours.
- *Sensor Data:* Continuous measurements from sensors capturing variables like pressure, humidity, or air quality.

2. Discrete Time Series Data: Discrete time series data, on the other hand, consists of measurements or observations that are limited to specific values or categories. Unlike continuous data, discrete data does not have a continuous range of possible values but instead comprises distinct and separate data points. Common examples include:

- *Count Data:* Tracking the number of occurrences or events within a specific time period.
- *Categorical Data:* Classifying data into distinct categories or classes (e.g., customer segments, product types).
- *Binary Data:* Recording data with only two possible outcomes or states.

Basic Time Series Concepts

- **Trend:** A trend represents the general direction in which a time series is moving over an extended period. It indicates whether the values are increasing, decreasing, or staying relatively constant.
- **Seasonality:** Seasonality refers to recurring patterns or cycles that occur at regular intervals within a time series, often corresponding to specific time units like days, weeks, months, or seasons.
- **Moving average:** The moving average method is a common technique used in time series analysis to smooth out short-term fluctuations and highlight longer-term trends or patterns in the data. It involves calculating the average of a set of consecutive data points, referred to as a "window" or "rolling window," as it moves through the time series.
- **Noise:** Noise, or random fluctuations, represents the irregular and unpredictable components in a time series that do not follow a discernible pattern. It introduces variability that is not attributable to the underlying trend or seasonality.
- **Differencing:** Differencing is used to make the difference in values of a specified interval. By default, it's one, we can specify different values for plots. It is the most popular method to remove trends in the data.
- **Stationarity:** A stationary time series is one whose statistical properties, such as mean, variance, and autocorrelation, remain constant over time.
- **Order:** The order of differencing refers to the number of times the time series data needs to be differenced to achieve stationarity.
- **Autocorrelation:** Autocorrelation, is a statistical method used in time series analysis to quantify

the degree of similarity between a time series and a lagged version of itself.

- **Resampling:** Resampling is a technique in time series analysis that involves changing the frequency of the data observations. It's often used to transform the data to a different frequency (e.g., from daily to monthly) to reveal patterns or trends more clearly.

1. Display the date and time information in different formats.

Importing the Libraries

We will import all the libraries that we will be using throughout this article in one place so that do not have to import every time we use it this will save both our time and effort.

- **Numpy** – A Python library that is used for numerical mathematical computation and handling multidimensional ndarray, it also has a very large collection of mathematical functions to operate on this array.
- **Pandas** – A Python library built on top of NumPy for effective matrix multiplication and dataframe manipulation, it is also used for data cleaning, data merging, data reshaping, and data aggregation.
- **Matplotlib** – It is used for plotting 2D and 3D visualization plots, it also supports a variety of output formats including graphs for data.

```
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
from statsmodels.graphics.tsaplots import plot_acf
from statsmodels.tsa.stattools import adfuller
```

Loading The Dataset

To load the dataset into a dataframe we will use the pandas read_csv() function. We will use head() function to print the first five rows of the dataset. Here we will use the '**parse_dates**' parameter in the read_csv function to convert the 'Date' column to the DatetimeIndex format. By default, Dates are stored in string format which is not the right format for time series data analysis.

```
# reading the dataset using read_csv
df = pd.read_csv("stock_data.csv",
                 parse_dates=True,
                 index_col="Date")

# displaying the first five rows of dataset
df.head()
```

Output:

	Unnamed: 0	Open	High	Low	Close	Volume	Name
Date							
2006-01-03	NaN	39.69	41.22	38.79	40.91	24232729	AABA
2006-01-04	NaN	41.22	41.90	40.77	40.97	20553479	AABA
2006-01-05	NaN	40.93	41.73	40.85	41.53	12829610	AABA
2006-01-06	NaN	42.88	43.57	42.80	43.21	29422828	AABA
2006-01-09	NaN	43.10	43.66	42.82	43.42	16268338	AABA

Dropping Unwanted Columns

We will drop columns from the dataset that are not important for our visualization.

```
# deleting column
df.drop(columns='Unnamed: 0', inplace =True)
df.head()
```

Output:

	Open	High	Low	Close	Volume	Name
Date						

```

2006-01-03 39.69 41.22 38.79 40.91 24232729 AABA
2006-01-04 41.22 41.90 40.77 40.97 20553479 AABA
2006-01-05 40.93 41.73 40.85 41.53 12829610 AABA
2006-01-06 42.88 43.57 42.80 43.21 29422828 AABA
2006-01-09 43.10 43.66 42.82 43.42 16268338 AABA

```

Plotting Line plot for Time Series data:

Since, the volume column is of continuous data type, we will use line graph to visualize it.

```

# Assuming df is your DataFrame
sns.set(style="whitegrid") # Setting the style to whitegrid for a clean background

plt.figure(figsize=(12, 6)) # Setting the figure size
sns.lineplot(data=df, x='Date', y='High', label='High Price', color='blue')

# Adding labels and title
plt.xlabel('Date')
plt.ylabel('High')
plt.title('Share Highest Price Over Time')
plt.show()

```

Output:



Resampling

To better understand the trend of the data we will use the **resampling method**, resampling the data on a monthly basis can provide a clearer view of trends and patterns, especially when we are dealing with daily data.

```

# Assuming df is your DataFrame with a datetime index
df_resampled = df.resample('M').mean() # Resampling to monthly frequency, using mean as an
aggregation function

sns.set(style="whitegrid") # Setting the style to whitegrid for a clean background

# Plotting the 'high' column with seaborn, setting x as the resampled 'Date'
plt.figure(figsize=(12, 6)) # Setting the figure size

```



```
sns.lineplot(data=df_resampled, x=df_resampled.index, y='High', label='Month Wise Average High Price', color='blue')

# Adding labels and title
plt.xlabel('Date (Monthly)')
plt.ylabel('High')
plt.title('Monthly Resampling Highest Price Over Time')
plt.show()
```

Output:



An Example of Time-Series Analysis with Python

Plotting Data Using Pyplot

Python brings a host of benefits to the table regarding time-series analysis:

- It is a user-friendly language.
- It is widely available in the open-source world.
- It has extensive library support.
- It can reuse existing code.

Python offers extensive specialized libraries and tools specifically designed for time-series analysis. These libraries, such as pandas, NumPy, statsmodels, and scikit-learn, provide various functions and tools tailored to the unique challenges of working with time-dependent data. They simplify complex operations, allowing you to focus on extracting meaningful insights rather than reinventing the wheel.

One of the numerous ways software engineers add value to an org is by performing **time-series analysis**. This powerful technique allows us to extract valuable insights from temporal data and consists in analyzing and making predictions based on time-based patterns

Python has quickly emerged as a preferred tool for data analysis due to its simplicity, versatility, and vast community support. With its intuitive syntax and extensive library ecosystem, this elegant programming language allows you to tackle complex problems efficiently.

Whether you are building a data-intensive application or working with an experienced data

scientist, Python provides a robust platform for exploring, visualizing, and modeling time-dependent data.

Let's see how Python can empower your work with time-series data. Consider the following example code snippet that loads a time-series dataset using pandas and plots it using Matplotlib:

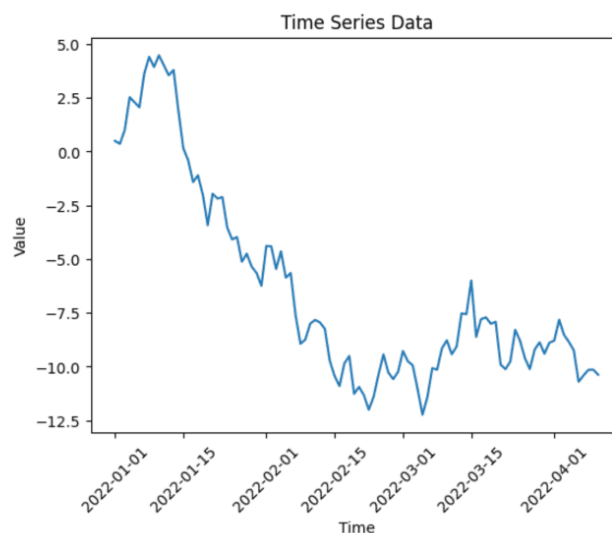
```
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np

# Generate random time-series data
np.random.seed(42)
dates = pd.date_range(start='2022-01-01', periods=100, freq='D')
values = np.random.randn(100).cumsum()

# Create a DataFrame from the generated data
data = pd.DataFrame({'date': dates, 'value': values})

# Set the 'date' column as the index
data.set_index('date', inplace=True)

# Plot the time-series data
plt.plot(data.index, data['value'])
plt.xlabel('Time')
plt.ylabel('Value')
plt.xticks(rotation = 45)
plt.title('Time Series Data')
plt.show()
```



This example consists of random data generated by NumPy's random number generator. The dataset consists of 100 dates, starting from January 1, 2022, and corresponding random values. The data is converted into a Pandas DataFrame, and the **'date'** column is set as the index. Finally, the time-series data is plotted using Matplotlib, displaying the variation of the **'value'** over time.

Working With Time Series in Python

Page | 58

Working with time-series data in Python involves several key steps, from choosing the right time-series library to loading and analyzing the data. Let's explore the essential aspects of working with

time series in Python, such as selecting a time-series library, utilizing the core library pandas for data loading, analysis, and visualization, and exploring some more specialized libraries for advanced time-series tasks.

Choosing a time-series library

Python provides various libraries tailored for time-series analysis. The core library for time-series analysis in Python is pandas. **Pandas** provides efficient data structures and functions to handle time series effectively. It allows you to load data from diverse sources, such as **CSV** files and databases like **Timescale**.

With pandas, you can perform basic analysis and visualization of time-series data. The central data structure in pandas is the DataFrame, which serves as the primary unit for representing time-series data.

Here's an example that demonstrates the steps of loading and working with time-series data using pandas in Python:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

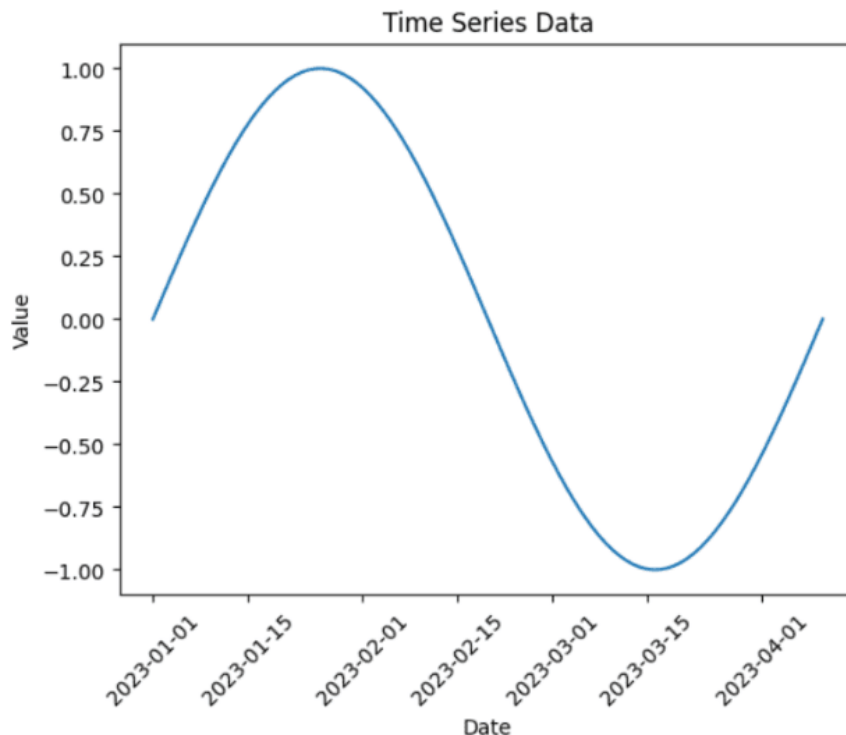
# Step 1: Load time-series Data
dates = pd.date_range(start='2023-01-01', periods=100)
values = np.sin(np.linspace(0, 2*np.pi, 100))
data = pd.DataFrame({'Date': dates, 'Value': values})

# Step 2: Perform Data Analysis
# Calculate summary statistics
summary_stats = data.describe()

# Filter data based on specific conditions
filtered_data = data[data['Value'] > 0]

# Resample data to a different frequency
resampled_data = data.resample('1W', on='Date').sum()

# Step 3: Visualize time-series Data
plt.plot(data['Date'], data['Value'])
plt.xlabel('Date')
plt.ylabel('Value')
plt.xticks(rotation = 45)
plt.title('Time Series Data')
plt.show()
```



This code generates a time-series dataset with dates and sine wave values. It performs data analysis tasks such as calculating summary statistics, filtering data based on conditions, and resampling the data to a different frequency. Finally, it visualizes the time-series data by plotting the values against the dates.

Try

1. Write a program and load a time series dataset with irregular time intervals (timestamps) into a pandas dataframe. Resample the data to have a regular time interval (e.g., daily, hourly) and fill any missing data with appropriate values (e.g., forward fill, backward fill, interpolation) on the following data:

```
data = {'Timestamp': ['2022-01-01 08:00:00', '2022-01-01 10:30:00', '2022-01-02 09:15:00'], 'Value': [10, 15, 20]}
```
2. Write a program and load above time series dataset that includes time zone information. Convert the timestamps to a common time zone (e.g., UTC) in pandas.
3. Write a program and load a time series dataset that exhibits seasonality (e.g., monthly sales data). Create additional columns to represent the year, quarter, month, day of the week, or any other seasonal components to facilitate seasonal analysis on the following data:

```
data = {'Date': ['2022-01-15', '2022-02-20', '2022-03-10', '2022-04-05', '2022-05-18'], 'Sales': [1000, 1200, 800, 1100, 1500]}
```

2. Generate summary statistics during a period.

Time-series data is a sequence of data points collected or recorded at successive points in time, often at regular intervals. Examples include stock prices, weather data, and server logs. Handling such data involves several key steps to extract meaningful insights.

1. Loading and Parsing Time-Series Data

- Time-series data typically includes a date or time column that must be converted into a datetime format.
- Using pandas, the `parse_dates` parameter ensures the date column is recognized as a datetime object, enabling time-based indexing.

Page | 60

2. Exploring the Data

- Inspect the dataset using `.head()`, `.info()`, and `.describe()` to understand its structure and basic

statistics.

- Check for missing values and handle them appropriately using methods like forward-fill (ffill) or backward-fill (bfill).

3. Filtering Data for a Specific Period

- Time-based indexing allows slicing the data for a specified range
- This extracts data between January 1, 2023, and June 30, 2023.

4. Generating Summary Statistics

Summary statistics provide a quick overview of the dataset's key characteristics:

- **Count:** Number of observations.
- **Mean:** Average value.
- **Standard Deviation (std):** Measure of data spread.
- **Min and Max:** Smallest and largest values.
- **Percentiles (25%, 50%, 75%):** Data distribution quartiles.

5. Optional Visualization

- Plotting the data helps in understanding trends, seasonality, or anomalies visually.

Summary Statistics for a Period:

- **Focused Analysis:** Restricting to a specific period helps analyze trends, patterns, or anomalies during that timeframe.
- **Decision-Making:** Insights from summary statistics aid in informed decision-making, such as planning resources or forecasting.
- **Comparisons:** Comparing statistics across different periods provides insights into changes over time.

Applications

1. **Finance:** Analyzing stock price movements over a quarter.
2. **Weather:** Evaluating seasonal temperature and rainfall statistics.
3. **Operations:** Studying server performance metrics during peak hours.

Challenges

1. **Irregular Intervals:** Missing or irregular timestamps require interpolation or resampling.
2. **Large Datasets:** Efficient handling of large datasets needs optimization techniques.
3. **Seasonality and Trends:** Identifying and analyzing patterns like seasonality may require additional tools like Fourier Transforms or ARIMA models

Learn how to load, clean, and analyze time-series data by generating summary statistics for a specific time period.

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
data = pd.read_csv("your_timeseries_data.csv", parse_dates=['Date'], index_col='Date')
print(data.head())
print(data.info())
print(data.describe())

# Handling missing values (if any)
data = data.fillna(method='ffill')
start_date = '2023-01-01'
end_date = '2023-06-30'

filtered_data = data.loc[start_date:end_date]
print(filtered_data.head())
summary_stats = filtered_data.describe()
print(summary_stats)
```

```
print(filtered_data.median())
print(filtered_data.std())
filtered_data['Column_of_Interest'].plot(title="Time Series Data for Selected Period",
figsize=(10, 6))
plt.xlabel("Date")
plt.ylabel("Value")
plt.show()
filtered_data.to_csv("filtered_timeseries_data.csv")
summary_stats.to_csv("summary_statistics.csv")
```

Output:

Filtered Data (First 5 Rows):

Date	Temperature	Rainfall	Humidity
2023-01-01	22.48	0.51	48.82
2023-01-02	19.31	6.61	64.92
2023-01-03	23.24	0.02	60.95
2023-01-04	27.62	7.01	85.74
2023-01-05	18.83	0.09	58.12

Summary Statistics:

Statistic	Temperature	Rainfall	Humidity
Count	181	181	181
Mean	19.91	2.15	63.20
Std. Dev.	4.74	2.37	14.39
Min	6.90	0.02	40.55
25%	16.62	0.55	50.39
Median (50%)	20.03	1.40	62.30
75%	22.71	2.93	74.61
Max	33.60	16.34	89.84

Try:

1. Write a program that computes summary statistics (mean, median, std) of the 'value' column for a specific date range in a pandas DataFrame.
2. Write a program that computes the average 'value' for each day of the week (e.g., Monday, Tuesday).
3. Write a program to compute the 7-day rolling mean for the 'value' column in a pandas DataFrame.
4. Write a program that computes the monthly average of 'value' but only for values greater than 30.
5. Write a program to compute the mean and standard deviation for multiple columns (e.g., 'value1' and 'value2') over a monthly period.

3. Compute rolling mean and rolling std deviations and plot.

Time Series Plot is used to observe various trends in the dataset over a period of time. In such

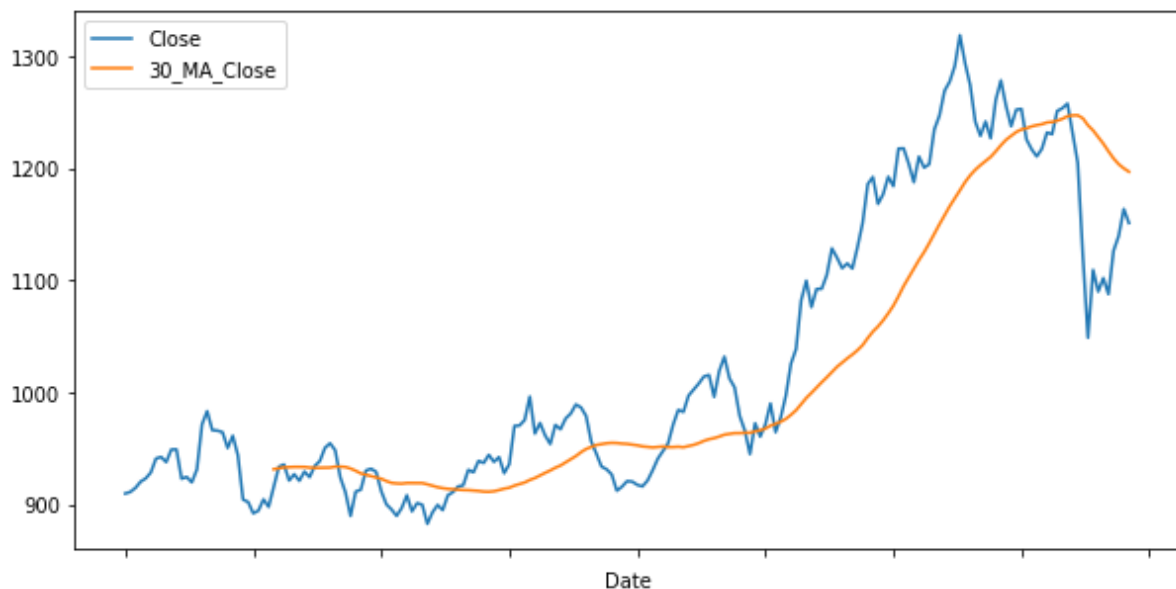
problems, the data is ordered by time and can fluctuate by the unit of time considered in the dataset (day, month, seconds, hours, etc.). When plotting the time series data, these fluctuations may prevent us to clearly gain insights about the peaks and troughs in the plot. So to clearly get value from the data, we use the rolling average concept to make the time series plot.

The rolling average or moving average is the simple mean of the last 'n' values. It can help us in finding trends that would be otherwise hard to detect. Also, they can be used to determine long-term trends. You can simply calculate the rolling average by summing up the previous 'n' values and dividing them by 'n' itself. But for this, the first (n-1) values of the rolling average would be Nan.

New kind of statistics: **rolling statistics**. Instead of computing a single statistic over an entire set of data, we compute a rolling statistic against a subset, or window, of that data, and we adjust the window with each new data point we encounter.

Pandas provides a number of functions to compute moving statistics. Given a DataFrame df and a window window, we can compute the rolling mean & rolling standard deviation of the columns in a DataFrame

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
%matplotlib inline
#importing data
df = pd.read_csv('Desktop\RELIANCE.NS.csv', index_col = 'Date')
df.tail()
#Calculating 30 days moving average
df['30_MA_Close'] = df['Close'].rolling(window=30).mean()
#calculating 20 days rolling standard deviation
df['20_std_Close'] = df['Close'].rolling(window=20).std()
df.head(31)
df[['Close', '30_MA_Close']].plot(figsize=(10,5))
```



Try:

1. Write a program that computes the **7-day rolling mean** and **7-day rolling standard deviation** for a 'value' column in a pandas DataFrame. Then, plot the original 'value' series along with the rolling mean and rolling standard deviation
2. Write a program to generate a DataFrame containing a time series of daily data for 60 days

with random values in the 'value' column.

3. Write a program to plots the original time series, the rolling mean, and the rolling standard deviation in a single graph with proper labels and legends."

7. Visualization of categorical data.

Visualization of categorical data refers to the use of charts and graphs to represent non-numerical data, often consisting of discrete groups or categories. These visualizations help in understanding the distribution, frequency, or comparison of categorical variables.

Key Concepts

1. Categorical Data:

- Data that represents distinct groups or categories.
- Examples: Gender (Male, Female), Product Types (Electronics, Furniture), Regions (North, South).

2. Types of Categorical Data:

- **Nominal:** Categories without any inherent order (e.g., Colors: Red, Blue, Green).
- **Ordinal:** Categories with a meaningful order but no fixed difference between values (e.g., Ratings: Poor, Fair, Good).

Common Visualization Techniques

- **Bar Charts:**
 - Categories with taller bars are more frequent or have higher values.
 - Useful for direct comparisons.
- **Grouped Bar Charts:**
 - Subcategories side-by-side allow insights into relationships within and between categories.
- **Stacked Bar Charts:**
 - Highlights total contributions and proportions simultaneously.
- **Pie Charts:**
 - Easier to interpret proportions but not ideal for precise comparisons.
- **Count Plots:**
 - Directly display the frequency of categories, useful for raw count analysis.

a. Plot categorical data as vertical and horizontal bar charts and label it.

A bar plot uses rectangular bars to represent data categories, with bar length or height proportional to their values. It compares discrete categories, with one axis for categories and the other for values.

Syntax: `plt.bar(x, height, width, bottom, align)`

Creating Vertical Bar Plots

For vertical bar plots, you can use the **bar()** function.

```
import matplotlib.pyplot as plt
import numpy as np

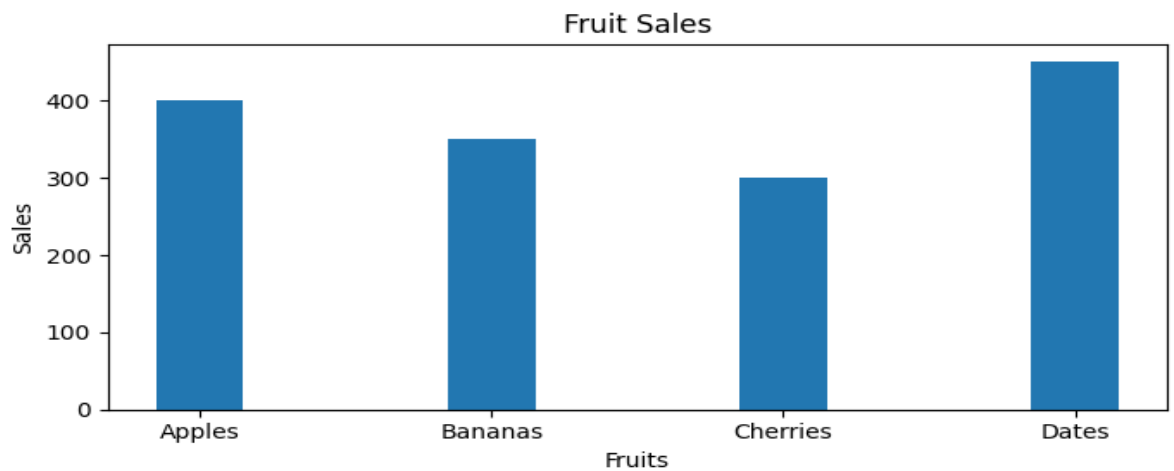
fruits = ['Apples', 'Bananas', 'Cherries', 'Dates']
sales = [400, 350, 300, 450]

plt.bar(fruits, sales, width=0.3)
plt.title('Fruit Sales')
plt.xlabel('Fruits')
plt.ylabel('Sales')
```



```
plt.show()
```

Output:



Vertical Plots

Creating Horizontal Bar Plots

For horizontal bar plots, you can use the **barh()** function. This function works similarly to **bar()**, but it displays bars horizontally:

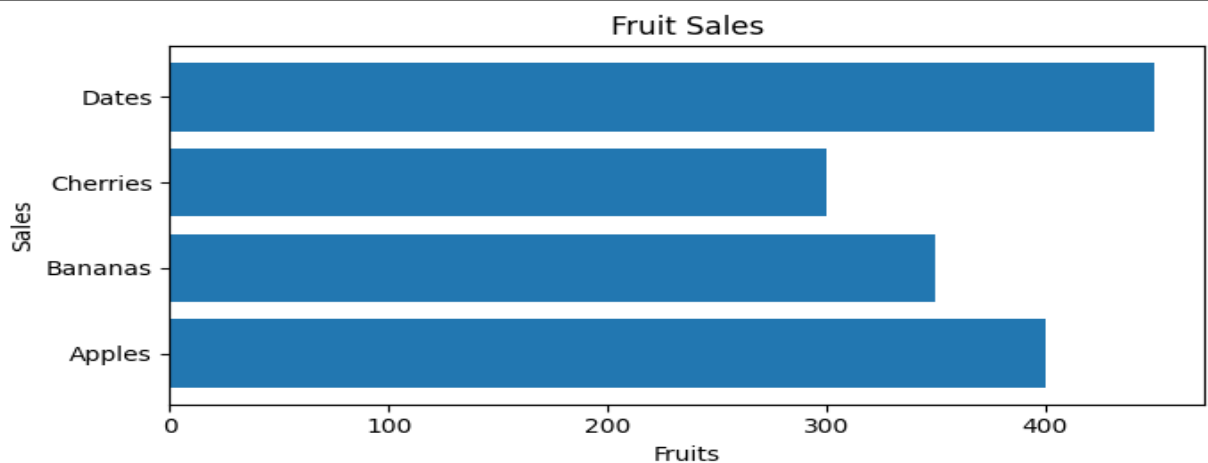
```
import matplotlib.pyplot as plt
import numpy as np

fruits = ['Apples', 'Bananas', 'Cherries', 'Dates']
sales = [400, 350, 300, 450]

plt.barh(fruits, sales)

plt.title('Fruit Sales')
plt.xlabel('Fruits')
plt.ylabel('Sales')
plt.show()
```

Output:



Horizontal Plots

Try:

- 1 Write a program to generate a DataFrame with categorical data representing the number of

- products sold in different categories (e.g., Electronics, Clothing, Food, etc.) for a particular week.
- 2 Write a program to Plots the data as: **vertical bar chart** showing the number of products sold for each category and **horizontal bar chart** showing the same data.
 - 3 Write a program to Both charts should be labeled with the category names on the x-axis (for the vertical bar chart) and y-axis (for the horizontal bar chart) Include a title and axis labels for both charts

b. Plot categorical data as vertical grouped bar charts and label it.

To Create a grouped bar plot in [Matplotlib](#)

- Matplotlib is a tremendous visualization library in Python for 2D plots of arrays. Matplotlib may be a multi-platform data visualization library built on [NumPy](#) arrays and designed to figure with the broader SciPy stack. It had been introduced by John Hunter within the year 2002.
- A bar plot or bar graph may be a graph that represents the category of knowledge with rectangular bars with lengths and heights that's proportional to the values which they represent. The bar plots are often plotted horizontally or vertically.
- A bar chart is a great way to compare categorical data across one or two dimensions. More often than not, it's more interesting to compare values across two dimensions and for that, a grouped bar chart is needed.

Approach:

- 1 Import Library (Matplotlib)
- 2 Import / create data.
- 3 Plot the bars in the grouped manner.

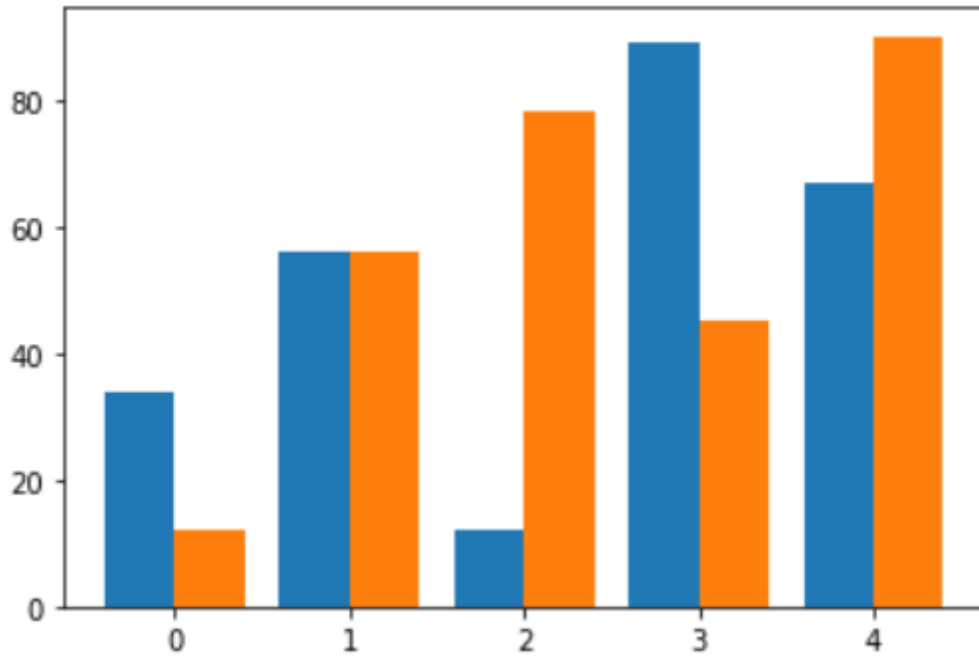
Example 1: (Simple grouped bar plot)

```
# importing package
import matplotlib.pyplot as plt
import numpy as np

# create data
x = np.arange(5)
y1 = [34, 56, 12, 89, 67]
y2 = [12, 56, 78, 45, 90]
width = 0.40

# plot data in grouped manner of bar type
plt.bar(x-0.2, y1, width)
plt.bar(x+0.2, y2, width)
```

Output:



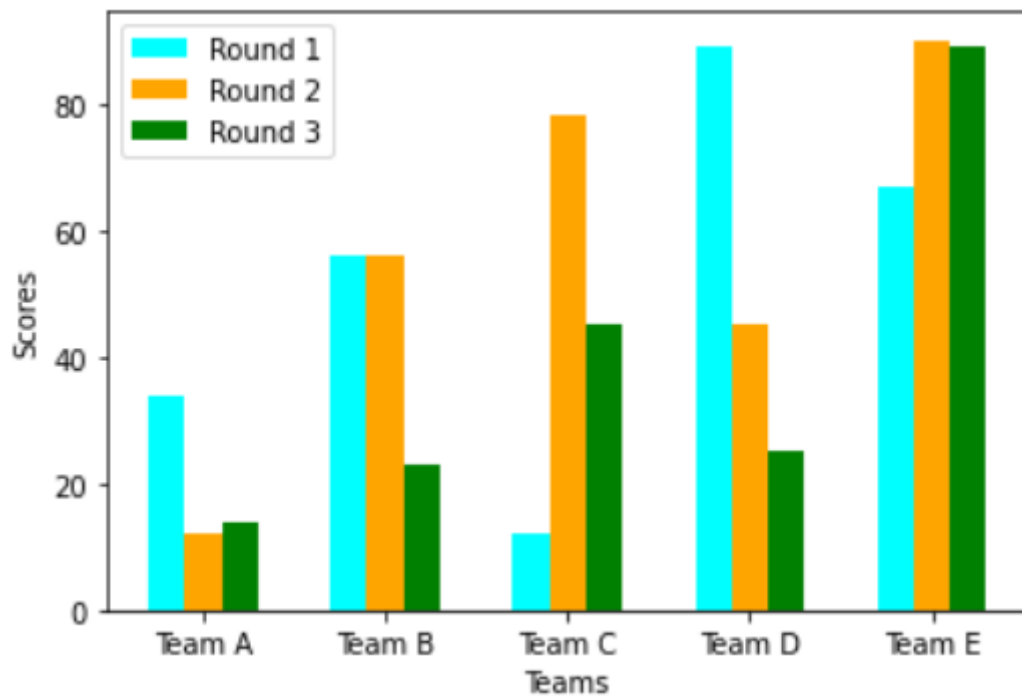
Example 2: (Grouped bar chart with more than 2 data)

```
# importing package
import matplotlib.pyplot as plt
import numpy as np

# create data
x = np.arange(5)
y1 = [34, 56, 12, 89, 67]
y2 = [12, 56, 78, 45, 90]
y3 = [14, 23, 45, 25, 89]
width = 0.2

# plot data in grouped manner of bar type
plt.bar(x-0.2, y1, width, color='cyan')
plt.bar(x, y2, width, color='orange')
plt.bar(x+0.2, y3, width, color='green')
plt.xticks(x, ['Team A', 'Team B', 'Team C', 'Team D', 'Team E'])
plt.xlabel("Teams")
plt.ylabel("Scores")
plt.legend(["Round 1", "Round 2", "Round 3"])
plt.show()
```

Output:



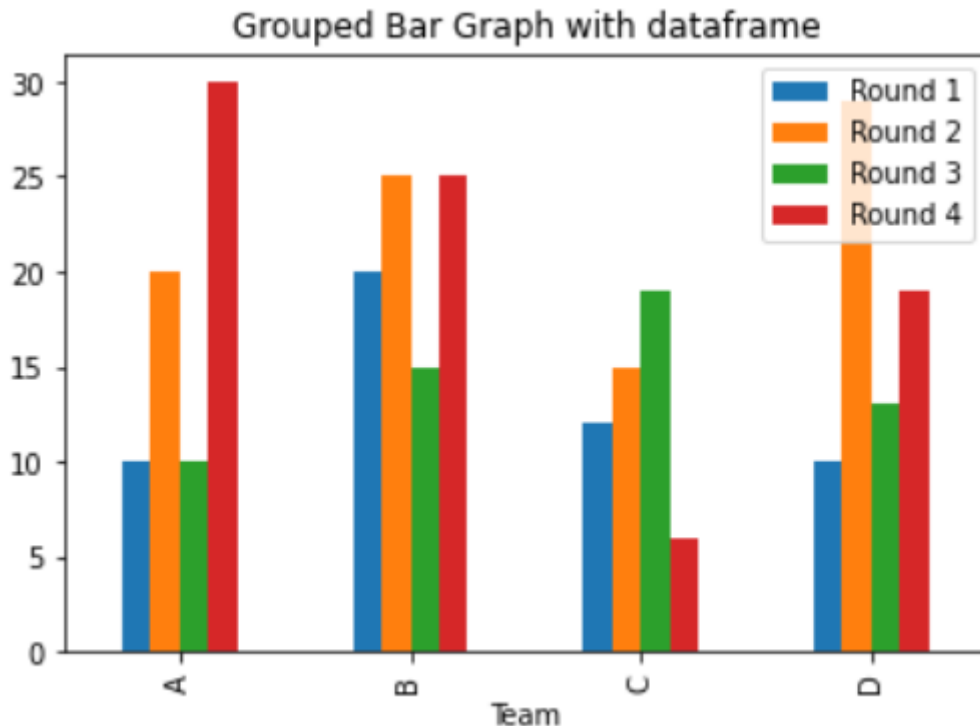
Example 3: (Grouped Bar chart using [dataframe](#) plot)

```
# importing package
import matplotlib.pyplot as plt
import pandas as pd

# create data
df = pd.DataFrame([[ 'A', 10, 20, 10, 30], [ 'B', 20, 25, 15, 25], [ 'C', 12, 15, 19, 6],
                  [ 'D', 10, 29, 13, 19]],
                  columns=[ 'Team', 'Round 1', 'Round 2', 'Round 3', 'Round 4'])

# view data
print(df)

# plot grouped bar chart
df.plot(x='Team',
        kind='bar',
        stacked=False,
        title='Grouped Bar Graph with dataframe')
```



Try:

- 1 Write a program to generate a DataFrame with categorical data representing the sales data of different product categories (e.g., Electronics, Clothing, Food) across two different time periods (e.g., January and February).
- 2 Write a program to plot the data as a **grouped vertical bar chart**, where the bars for each category (Electronics, Clothing, Food) are grouped side by side for the two months (January and February).
- 3 Write a program to ensure that the chart is properly labeled with the product categories on the x-axis, sales figures on the y-axis, and each group of bars representing the two months. Include a title and axis labels for the chart. Use different colors to distinguish the two months.

c. Plot categorical data as vertical stacked bar charts and label it.

To create a stacked bar plot in [Matplotlib](#).

- Matplotlib is a tremendous visualization library in Python for 2D plots of arrays. Matplotlib may be a multi-platform data visualization library built on [NumPy](#) arrays and designed to figure with the broader SciPy stack.
- A bar plot or bar graph may be a graph that represents the category of knowledge with rectangular bars with lengths and heights that are proportional to the values which they represent. The bar plots are often plotted horizontally or vertically.
- Stacked bar plots represent different groups on the highest of 1 another. The peak of the bar depends on the resulting height of the mixture of the results of the groups. It goes from rock bottom to the worth rather than going from zero to value.

Approach:

1. Import Library (Matplotlib)
 2. Import / create data.
1. Plot the bars in the stack manner.

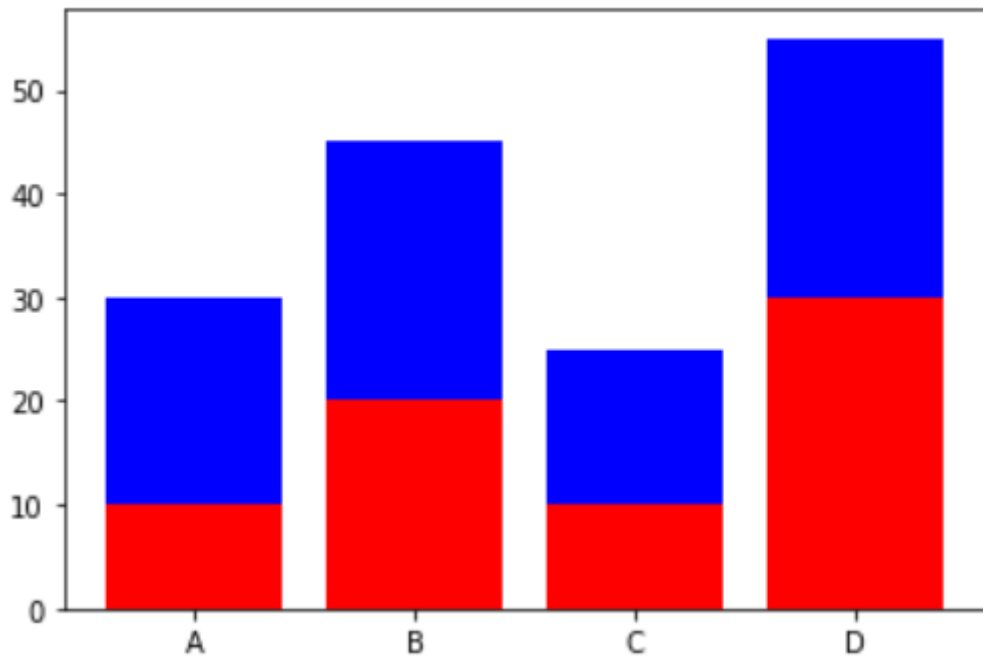
Example 1: (Simple stacked bar plot)

```
# importing package
import matplotlib.pyplot as plt
```

```
# create data
x = ['A', 'B', 'C', 'D']
y1 = [10, 20, 10, 30]
y2 = [20, 25, 15, 25]

# plot bars in stack manner
plt.bar(x, y1, color='r')
plt.bar(x, y2, bottom=y1, color='b')
plt.show()
```

Output:



Example 2: (Stacked bar chart with more than 2 data)

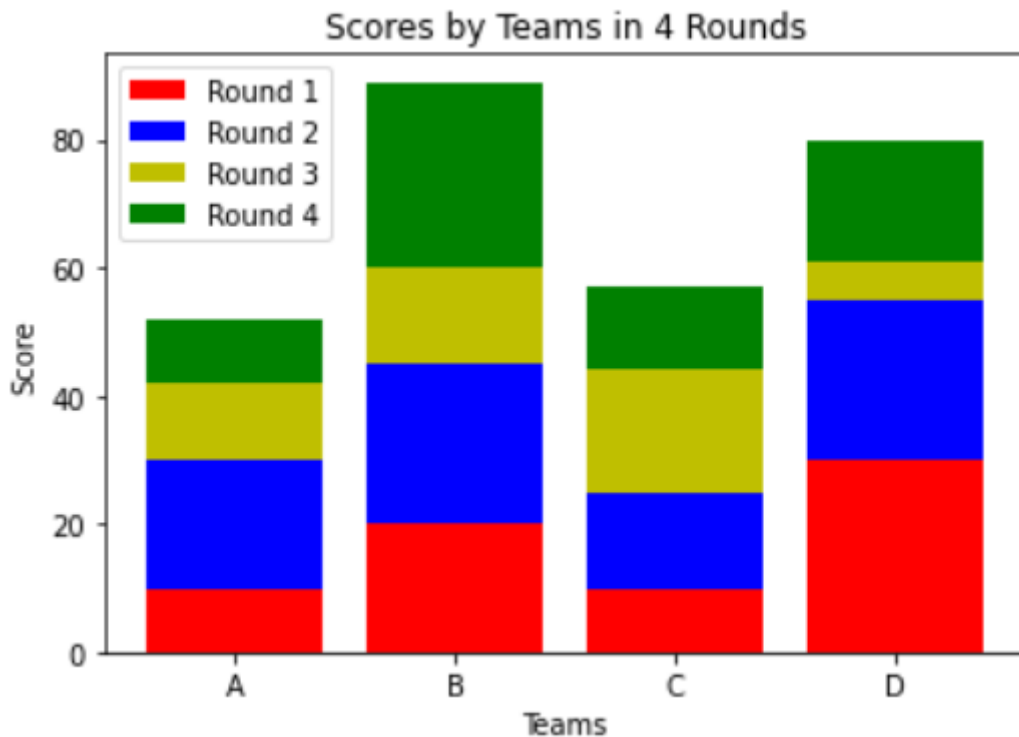
```
# importing package
import matplotlib.pyplot as plt
import numpy as np

# create data
x = ['A', 'B', 'C', 'D']
y1 = np.array([10, 20, 10, 30])
y2 = np.array([20, 25, 15, 25])
y3 = np.array([12, 15, 19, 6])
y4 = np.array([10, 29, 13, 19])

# plot bars in stack manner
plt.bar(x, y1, color='r')
plt.bar(x, y2, bottom=y1, color='b')
plt.bar(x, y3, bottom=y1+y2, color='y')
plt.bar(x, y4, bottom=y1+y2+y3, color='g')
plt.xlabel("Teams")
plt.ylabel("Score")
```

```
plt.legend(["Round 1", "Round 2", "Round 3", "Round 4"])
plt.title("Scores by Teams in 4 Rounds")
plt.show()
```

Output:



Example 3: (Stacked Bar chart using [dataframe plot](#)):

```
# importing package
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd

# create data
df = pd.DataFrame([[ 'A', 10, 20, 10, 26], [ 'B', 20, 25, 15, 21], [ 'C', 12, 15, 19, 6],
                  [ 'D', 10, 18, 11, 19]],
                  columns=[ 'Team', 'Round 1', 'Round 2', 'Round 3', 'Round 4'])

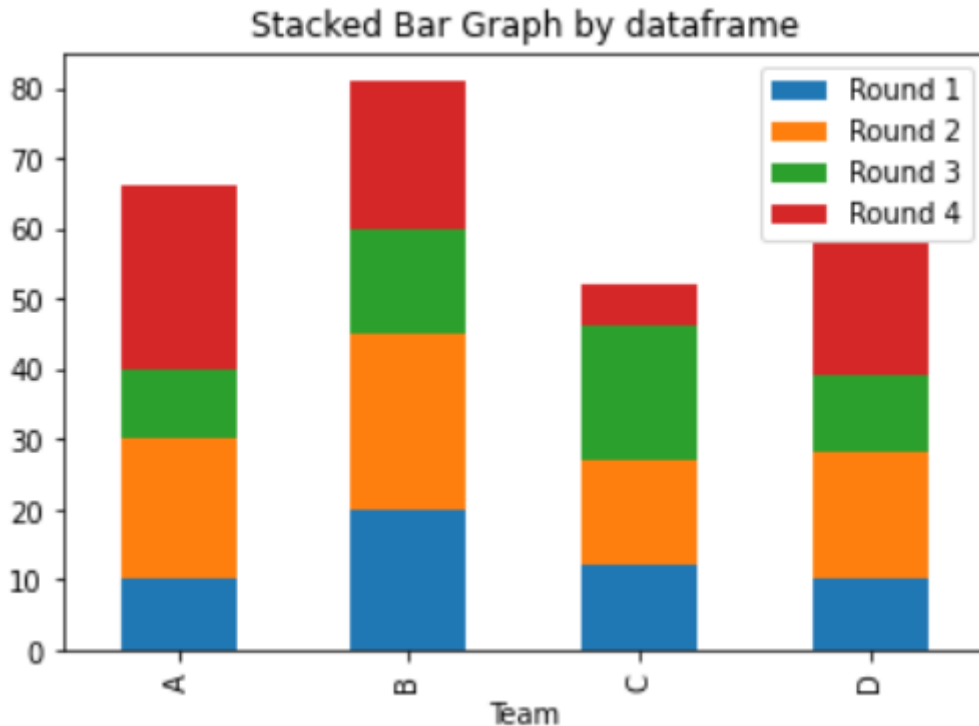
# view data
print(df)

# plot data in stack manner of bar type
df.plot(x='Team', kind='bar', stacked=True,
        title='Stacked Bar Graph by dataframe')

plt.show()
```

Output :

	Team	Round 1	Round 2	Round 3	Round 4
0	A	10	20	10	26
1	B	20	25	15	21
2	C	12	15	19	6
3	D	10	18	11	19



Try:

1. Write a program to generate a DataFrame with categorical data representing the sales data of different product categories (e.g., Electronics, Clothing, Food) across two different regions (e.g., North and South).
2. Write a program to plot the data as a stacked vertical bar chart, where the bars for each product category represent the sales data from the two regions stacked on top of each other.
3. Write a program to ensure that the chart is properly labeled with the product categories on the x-axis, sales figures on the y-axis, and each stacked section representing a different region. Include a title and axis labels for the chart. Use different colors for each region to differentiate them in the stack.

d. Interpret the results.

Vertical and Horizontal Bar Charts:

- Vertical and horizontal bar charts show the individual value of each category.

Example Interpretation:

- In the example, category B has the highest value, while category D has the lowest. Horizontal bar charts are especially useful when category labels are long.

Vertical Grouped Bar Chart:

- Displays subcategories (e.g., Group 1 and Group 2) side-by-side for each main category.

Example Interpretation:

- In the grouped chart, for category A, Group 2 has a slightly higher value than Group 1. It is easy to compare subcategories within each main category and between categories.

Vertical Stacked Bar Chart:

- Shows the total value for each category, with different colors indicating subcategory contributions.

Example Interpretation:

- The stacked bar chart highlights the total contribution of Group 1 and Group 2 to each category. For category C, Group 1 contributes a larger proportion than Group 2.

8. Visualization of correlations.

A correlation describes the relationship between two variables. If an increase in one variable produces an increase in the other one, that's a **positive correlation**. If an increase in one variable results in a decrease in the other, that's a **negative correlation**.

There are several different correlation coefficients, but the most popular one is **Pearson's correlation** (a.k.a Pearson's R). If someone mentions a correlation without specifying which coefficient they use, then most probably they use the Pearson's R. We'll use it in our topic too. One important thing — Pearson's correlation is for numeric data only. Techniques for locating associations in categorical data are more advanced.

Range	Meaning
0.70.7 to 1.01.0	a strong positive correlation
0.30.3 to 0.70.7	a weak positive correlation
-0.3-0.3 to 0.30.3	a negligible correlation
-0.7-0.7 to -0.3-0.3	a weak negative correlation
-1.0-1.0 to -0.7-0.7	a strong negative correlation

1. Plot the pair wise scatter plots of numerical attributes.

Data Visualization is the presentation of data in pictorial format. It is extremely important for Data Analysis, primarily because of the fantastic ecosystem of data-centric Python packages. **Seaborn** is one of those packages that can make analyzing data much easier.

Pairplot Seaborn to analyze data and, using the `sns.pairplot()` function

PairPlot Seaborn : Implementation

1. Pairplot Seaborn: Plotting Selected Variables
2. Pairplot Seaborn: Adding a Hue Color to a Seaborn Pairplot
3. Pairplot Seaborn: Modifying Color Palette
4. Pairplot Seaborn: Diagonal Kind of plots
5. Pairplot Seaborn: Adjusting Plot Kind
6. Pairplot Seaborn: Controlling the Markers
7. Pairplot Seaborn: Limiting the Variables

PairPlot Seaborn : Implementation

To implement a Pair Plot using Seaborn, you can follow these steps:

To plot multiple pairwise bivariate distributions in a dataset, you can use the `pairplot()` function. This shows the relationship for (n, 2) combination of variable in a DataFrame as a matrix of plots and the diagonal plots are the univariate plots.

Syntax: `seaborn.pairplot(data, **kwargs)`

Parameter:

data: Tidy (long-form) dataframe where each column is a variable and each row is an observation.

hue: Variable in "data" to map plot aspects to different colors.

palette: dict or seaborn color palette

{x, y}_vars: lists of variable names, optional

dropna: boolean, optional

First of all, We see Upload seaborn library 'tips' using pandas. Then, we will visualize data with seaborn.

```
# importing packages
import seaborn
import matplotlib.pyplot as plt
# loading dataset using seaborn
df = seaborn.load_dataset('tips')
df.head()
```

Output:

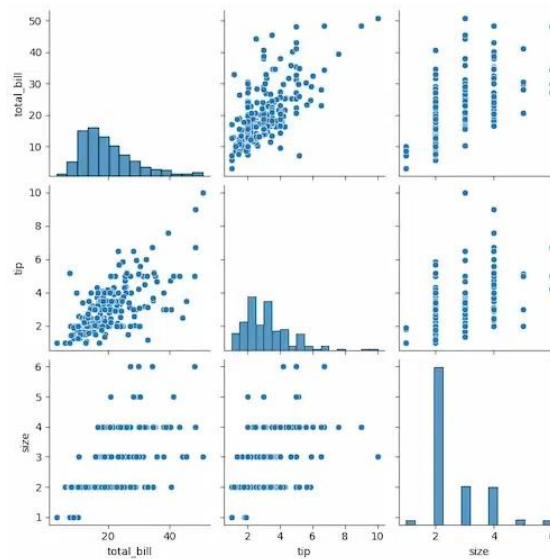
	total_bill	tip	sex	smoker	day	time	size
0	16.99	1.01	Female	No	Sun	Dinner	2
1	10.34	1.66	Male	No	Sun	Dinner	3
2	21.01	3.50	Male	No	Sun	Dinner	3
3	23.68	3.31	Male	No	Sun	Dinner	2
4	24.59	3.61	Female	No	Sun	Dinner	4

Let's plot pairplot using seaborn:

We will simply plot a pairplot with tips data frame.

```
seaborn.pairplot(df)
plt.show()
```

Output:



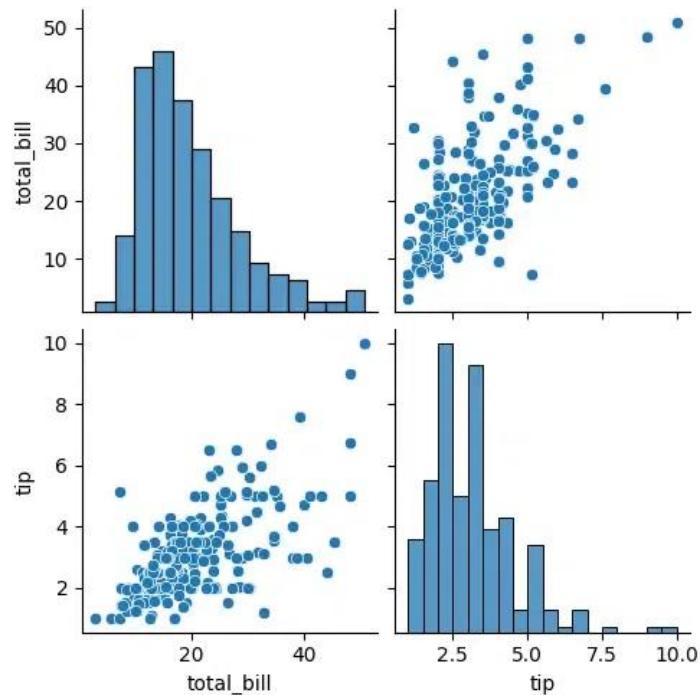
seaborn pairplot

1. Pairplot Seaborn: Plotting Selected Variables

```
import seaborn as sns
import matplotlib.pyplot as plt

df = sns.load_dataset('tips')
selected_vars = ['total_bill', 'tip']
sns.pairplot(df, vars=selected_vars)
plt.show()
```

Output:

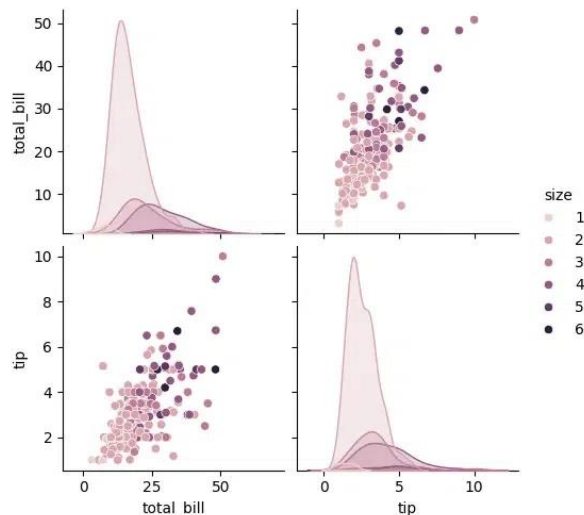


pairplot seaborn

2. Pairplot Seaborn: Adding a Hue Color to a Seaborn Pairplot

```
import seaborn
import matplotlib.pyplot as plt
df = seaborn.load_dataset('tips')
seaborn.pairplot(df, hue='size')
plt.show()
```

Output:



pairplot seaborn

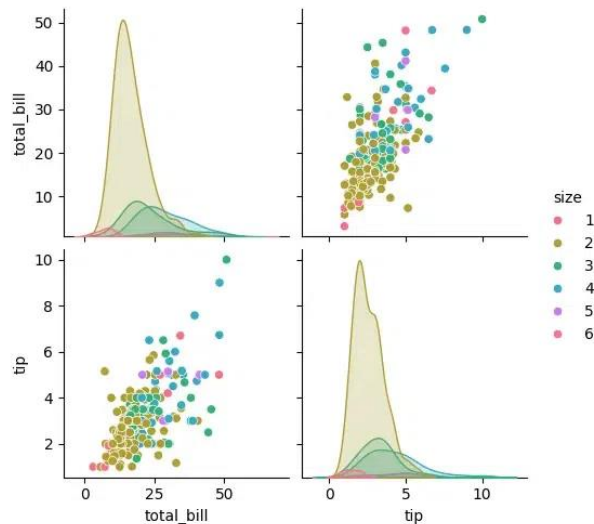
- The points in this scatter plot are colored by the value of size, so you can see how the relationship between total_bill and tip varies depending on the size of the party.
- There is a positive correlation between total_bill and tip. This means that, in general, larger bills tend to have larger tips
- There is a positive correlation between tip and size. This means that, in general, larger parties tend to have larger tips.
- The relationship between tip and size is stronger for larger total bill amounts.

3. Pairplot Seaborn: Modifying Color Palette

```
import seaborn as sns
import matplotlib.pyplot as plt

df = sns.load_dataset('tips')
sns.pairplot(df, hue="size", palette="husl")
plt.show
```

Output:



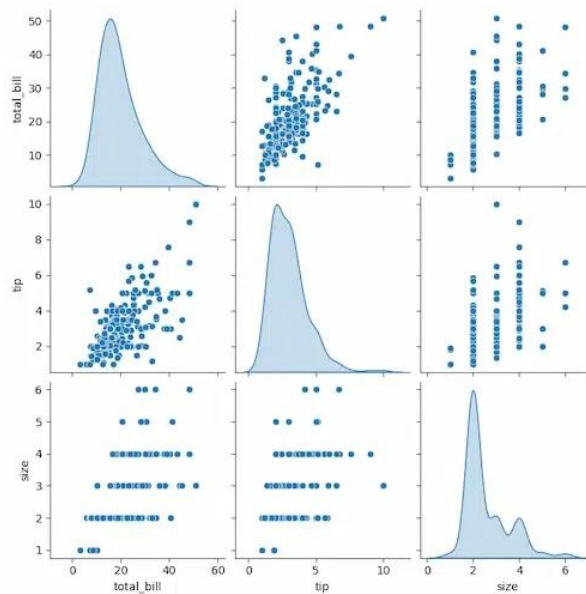
2. Pairplot Seaborn: Diagonal Kind of plots

In Seaborn's Pairplot, the 'diag_kind' parameter specifies the type of plot to display along the diagonal axis, representing the univariate distribution of each variable. Options include 'hist' for histograms, 'kde' for kernel density estimates, and 'scatter' for scatterplots. Choose based on the nature of the data and analysis goals. Here, let's plot with kernel density estimates.

```
import seaborn as sns
import matplotlib.pyplot as plt

df = sns.load_dataset('tips')
sns.pairplot(df, diag_kind = 'kde')
plt.show
```

Output:

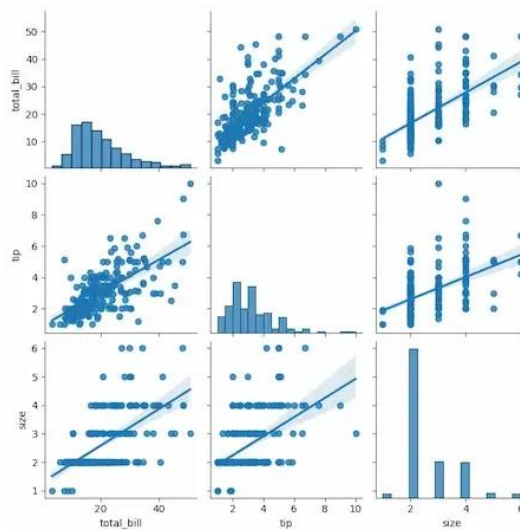


5. Pairplot Seaborn: Adjusting Plot Kind

The kind parameter allows to change the type of plot used for the off-diagonal plots. You can choose any like scatter, kde, or reg (regression).

```
sns.pairplot(df, kind='reg')
plt.show()
```

Output:



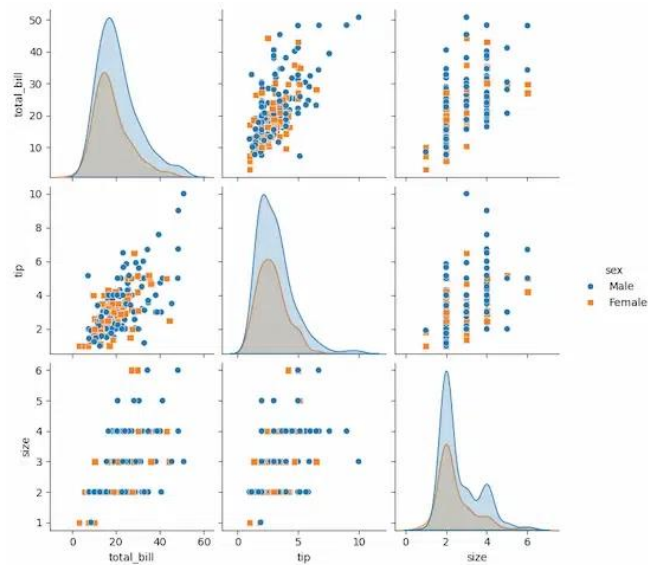
Adjusting Plot Kind

6. Pairplot Seaborn: Controlling the Markers

The markers parameter allows you to specify different markers for different categories.

```
sns.pairplot(df, hue='sex', markers=["o", "s"])
plt.show()
```

Output:



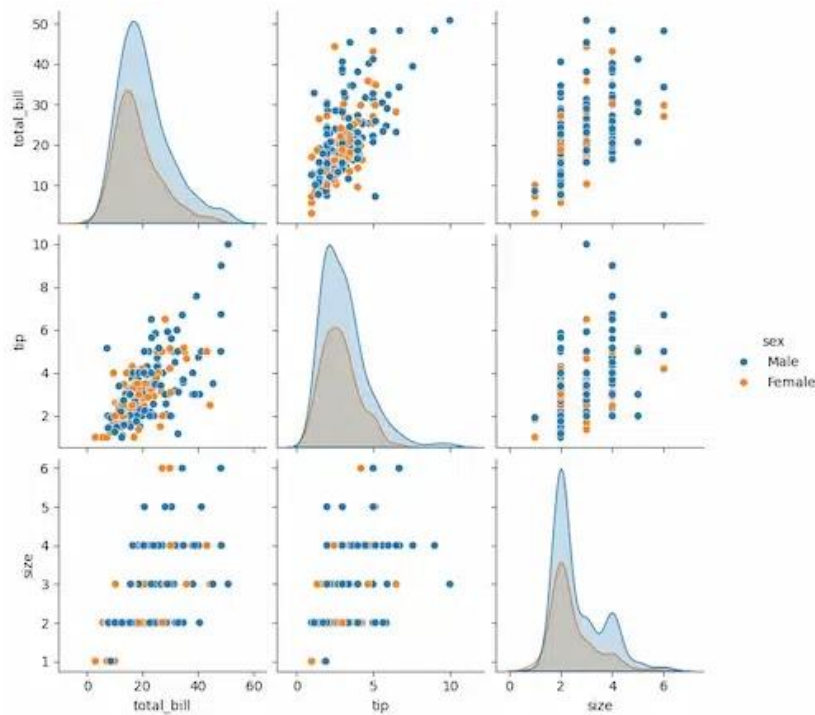
Controlling the Markers

7. Pairplot Seaborn: Limiting the Variables

If you are interested in only a subset of the variables, you can specify them using the `vars` parameter.

```
sns.pairplot(df, hue='sex', vars=['total_bill', 'tip', 'size'])
plt.show()
```

Output:



Pairplot Seaborn: Limiting the Variables

Try:

1. Write a program to generate a DataFrame containing numerical data for four attributes (e.g., Attribute1, Attribute2, Attribute3, Attribute4).

2. Write a program to plots the **pairwise scatter plots** of the numerical attributes to visualize the relationships between each pair of attributes.
3. Write a program to use a pair plot to display all possible scatter plots between the attributes in one plot, Include proper labels for the axes and a title for the plot.

2. Identify the type of correlations.

Definition

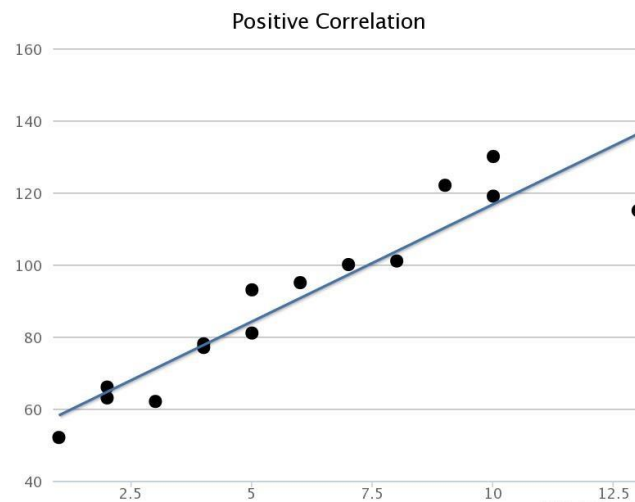
Correlation describes the relationship between [variables](#). It can be described as either strong or weak, and as either positive or negative.

Note: 1= Correlation does not imply causation.

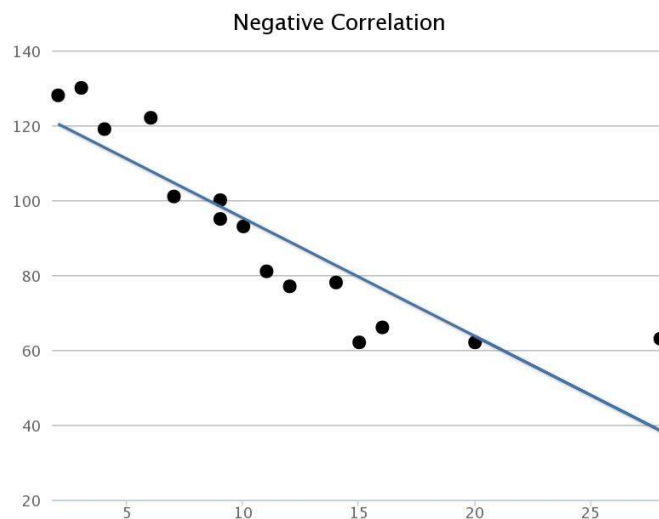
Types of Correlation

There are four types of correlation:

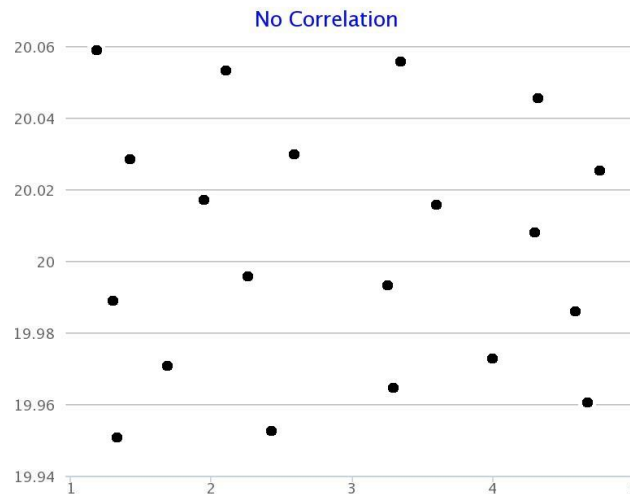
1. **Positive Correlation:** Positive correlation indicates that two variables have a direct relationship. As one variable increases, the other variable also increases. For example, there is a positive correlation between height and weight. As people get taller, they also tend to weigh more.



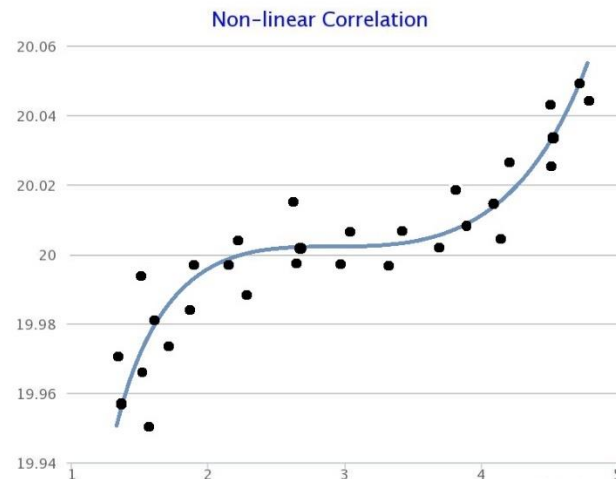
2. **Negative Correlation:** Negative correlation indicates that two variables have an inverse relationship. As one variable increases, the other variable decreases. For example, there is a negative correlation between price and demand. As the price of a product increases, the demand for that product decreases.



3. **No Correlation:** No correlation indicates that there is no relationship between two variables. The changes in one variable do not affect the other variable. For example, there is no correlation between shoe size and intelligence.



4. **Non-linear Correlation (known as curvilinear correlation):** There is a *non-linear correlation* when there is a relationship between variables but the relationship is not [linear](#) (straight).



Steps to Identify Correlations

1. Heatmap

- **Code Example:**

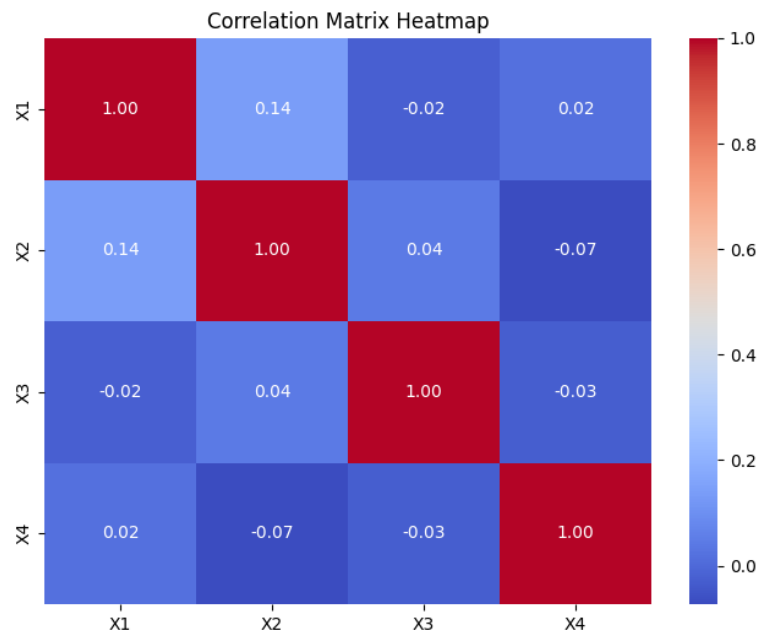
```
import seaborn as sns
import matplotlib.pyplot as plt

# Example data
import numpy as np
import pandas as pd
data = {
    'X1': np.random.rand(100),
    'X2': np.random.rand(100) * 2,
    'X3': np.random.rand(100) * 0.5 + np.linspace(0, 1, 100),
    'X4': np.random.rand(100)
}
df = pd.DataFrame(data)
```



```
# Correlation Matrix Heatmap
plt.figure(figsize=(8, 6))
sns.heatmap(df.corr(), annot=True, cmap='coolwarm', fmt=".2f")
plt.title('Correlation Matrix Heatmap')
plt.show()
```

Output:



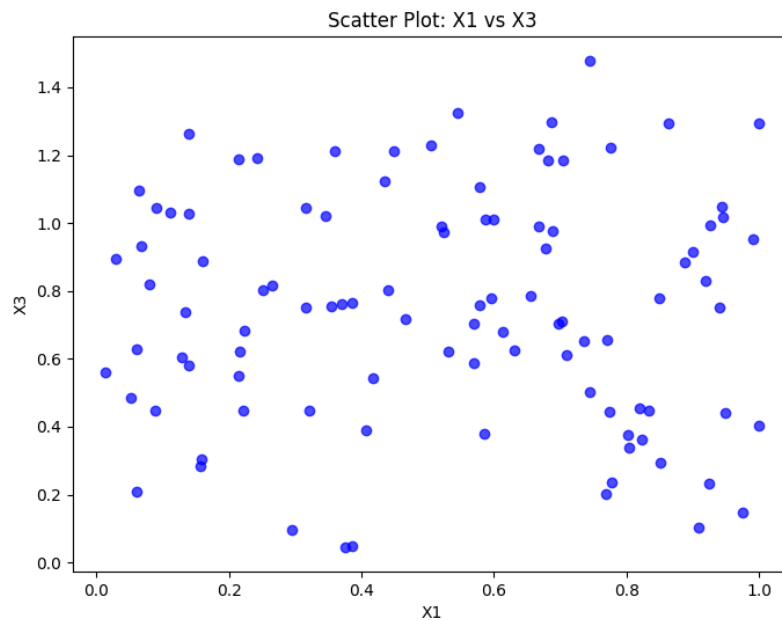
2. Scatter Plot

- Code Example:**

```
plt.figure(figsize=(8, 6))

# Scatter plot for a pair of variables
plt.scatter(df['X1'], df['X3'], alpha=0.7, color='blue')
plt.title('Scatter Plot: X1 vs X3')
plt.xlabel('X1')
plt.ylabel('X3')
plt.show()
```

Output:

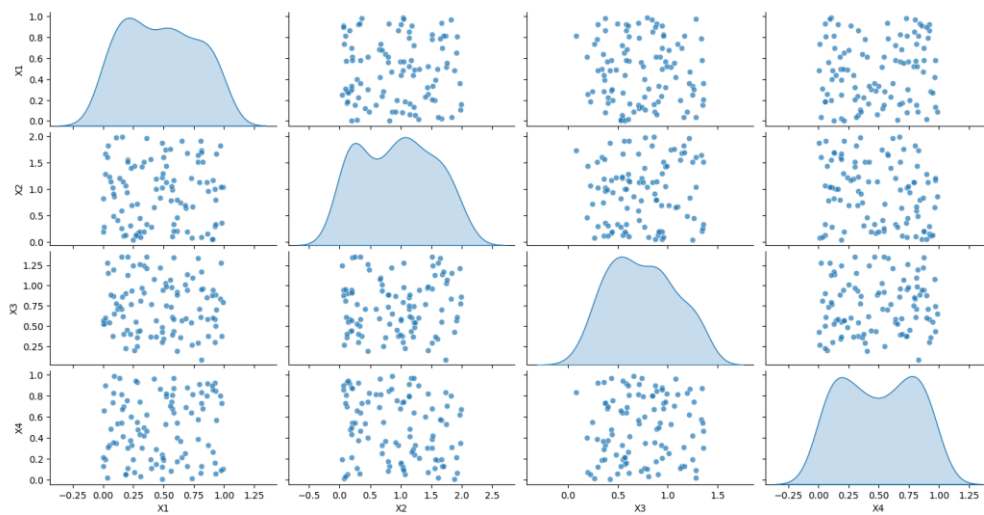


3. Pair Plot

- **Code Example:**

```
sns.pairplot(df, diag_kind='kde', plot_kws={'alpha': 0.7})
plt.suptitle('Pair Plot of Variables', y=1.02)
plt.show()
```

Output:

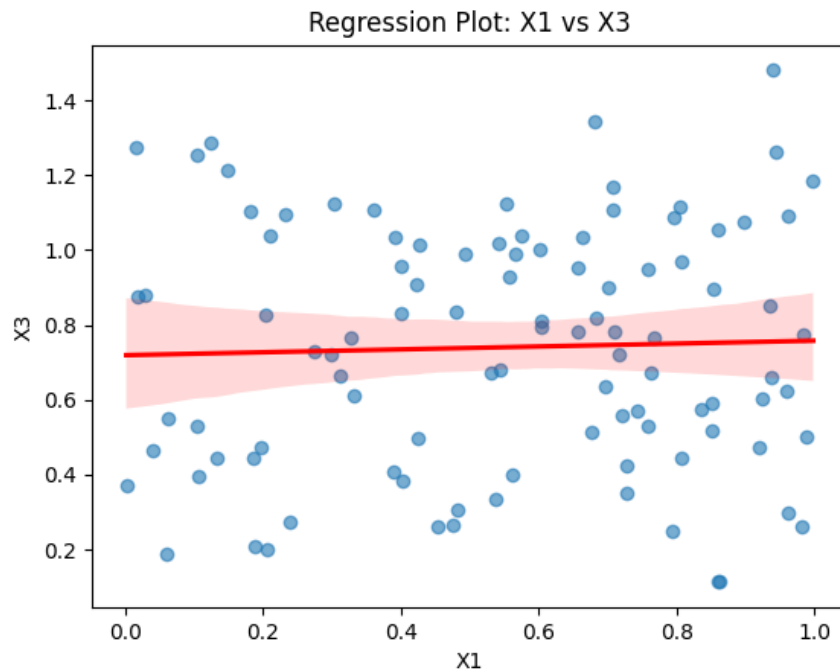


4. Regression Plot

- **Code Example:**

```
sns.regplot(x='X1', y='X3', data=df, scatter_kws={'alpha': 0.6}, line_kws={'color': 'red'})
plt.title('Regression Plot: X1 vs X3')
plt.xlabel('X1')
plt.ylabel('X3')
plt.show()
```

Output:



Try:

1. Write a program to Generates a DataFrame with numerical data for five attributes (e.g., Attribute1, Attribute2, Attribute3, Attribute4, Attribute5).
2. Write a program to computes the **correlation matrix** between the numerical attributes to identify the relationships between them.
3. Write a program to Identifies the type of correlation for each pair of attributes (i.e., **positive**, **negative**, or **no correlation**).
4. Write a program to Display the correlation matrix and print a summary of the type of correlation for each pair of attributes.

5. Interpret the results.

Interpretation of Correlation coefficients

- Perfect: 0.80 to 1.00
- Strong: 0.50 to 0.79
- Moderate: 0.30 to 0.49
- Weak: 0.00 to 0.29

Value greater than 0.7 is considered a strong correlation between variables.

Type	Visualization	Correlation Coefficient
Positive Correlation	Scatter plot: upward trend	$0 < r \leq 1$
Negative Correlation	Scatter plot: downward trend	$-1 \leq r < 0$

No Correlation	Scatter plot: no trend	$r \approx 0$
Non-Linear Correlation	Scatter plot: curved or complex patterns	Spearman or advanced tests

9. Visualization of distributions.

Data visualization building block is learning to summarize lists of factors or numeric vectors. More often than not, the best way to share or explore this summary is through data visualization. The most basic statistical summary of a list of objects or numbers is its distribution. Once a data has been summarized as a distribution, there are several data visualization techniques to effectively relay this information. For this reason, it is important to have a deep understand the concept of a distribution.

1. Plot the histograms of numerical data.

To create a Matplotlib histogram the first step is to create a bin of the ranges, then distribute the whole range of the values into a series of intervals, and count the values that fall into each of the intervals. Bins are identified as consecutive, non-overlapping intervals of variables. The `matplotlib.pyplot.hist()` function is used to compute and create a histogram of x

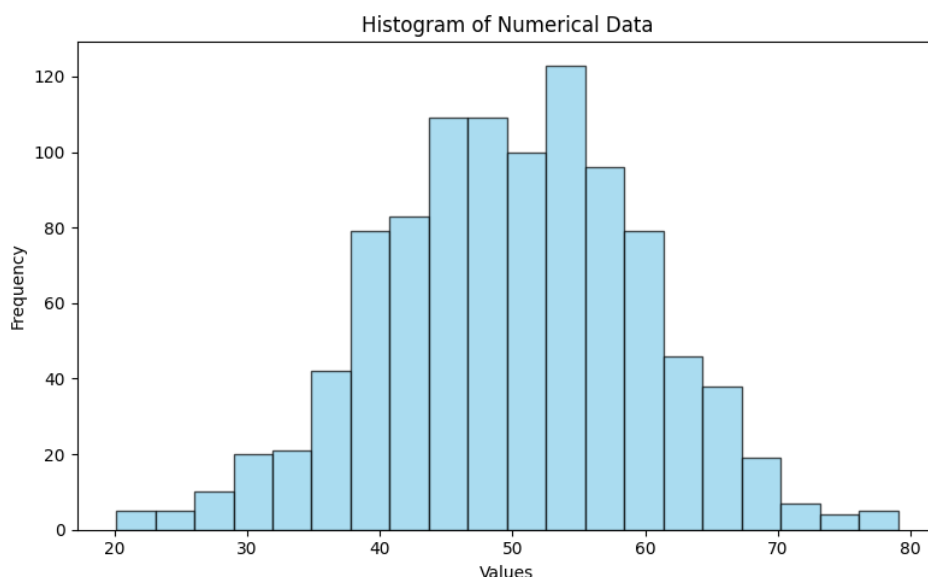
Code Example:

```
import numpy as np
import matplotlib.pyplot as plt

# Example numerical data
data = np.random.normal(loc=50, scale=10, size=1000) # Normal distribution

# Histogram
plt.figure(figsize=(8, 5))
plt.hist(data, bins=20, color='skyblue', edgecolor='black', alpha=0.7)
plt.title('Histogram of Numerical Data')
plt.xlabel('Values')
plt.ylabel('Frequency')
plt.tight_layout()
plt.show()
```

Output:



Try:

1. Write a program to plot a histogram with different Customization.

2. Write a program to plot a Stacked Histograms on the above data points.
3. Write a program to Generates a DataFrame with numerical data for five attributes (e.g., Attribute1, Attribute2, Attribute3, Attribute4, Attribute5).
4. Write a program to Plots the **histograms** of each numerical attribute to show the distribution of the data.
5. Write a program to Ensure that the histograms are clearly labeled with titles, axis labels, and a legend to differentiate between the attributes.

2. Plot the counts of categorial data.

To plot the count of categorial data, you can use a bar chart, which shows the distribution of a categorial variable by making the height of each bar proportional to the number of cases in each group:

Seaborn is an amazing visualization library for statistical graphics plotting in Python. It provides beautiful default styles and color palettes to make statistical plots more attractive. It is built on the top of [matplotlib](#) library and also closely integrated to the data structures from [pandas](#). **Seaborn.countplot()**

- o **seaborn.countplot()** method is used to Show the counts of observations in each categorial bin using bars.

*Syntax : seaborn.countplot(x=None, y=None, hue=None, data=None, order=None, hue_order=None, orient=None, color=None, palette=None, saturation=0.75, dodge=True, ax=None, **kwargs)*

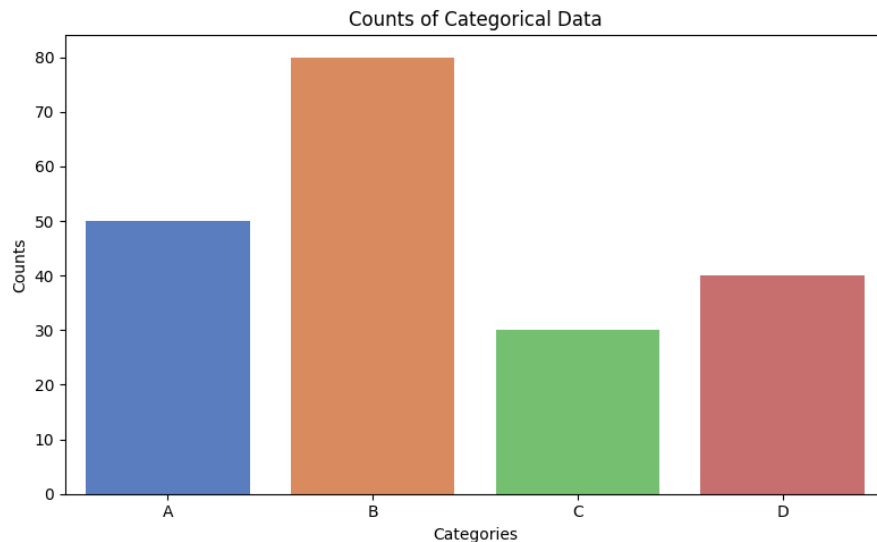
Code Example:

```
import seaborn as sns

# Example categorial data
categories = ['A', 'B', 'C', 'D']
counts = [50, 80, 30, 40]

# Bar plot for categorial data
plt.figure(figsize=(8, 5))
sns.barplot(x=categories, y=counts, palette='muted')
plt.title('Counts of Categorial Data')
plt.xlabel('Categories')
plt.ylabel('Counts')
plt.tight_layout()
plt.show()
```

Output:



Try:

1. Write a program to Generates a DataFrame with categorical data representing different product categories (e.g., Electronics, Clothing, Food, Toys, Books).
2. Write a program to Plots the **count of occurrences** for each category using a **bar chart**.
3. Write a program to Ensure that the chart is clearly labeled with titles, axis labels, and a legend to differentiate the categories.

3. Plot the data distributions (or densities).

Kernel density estimation

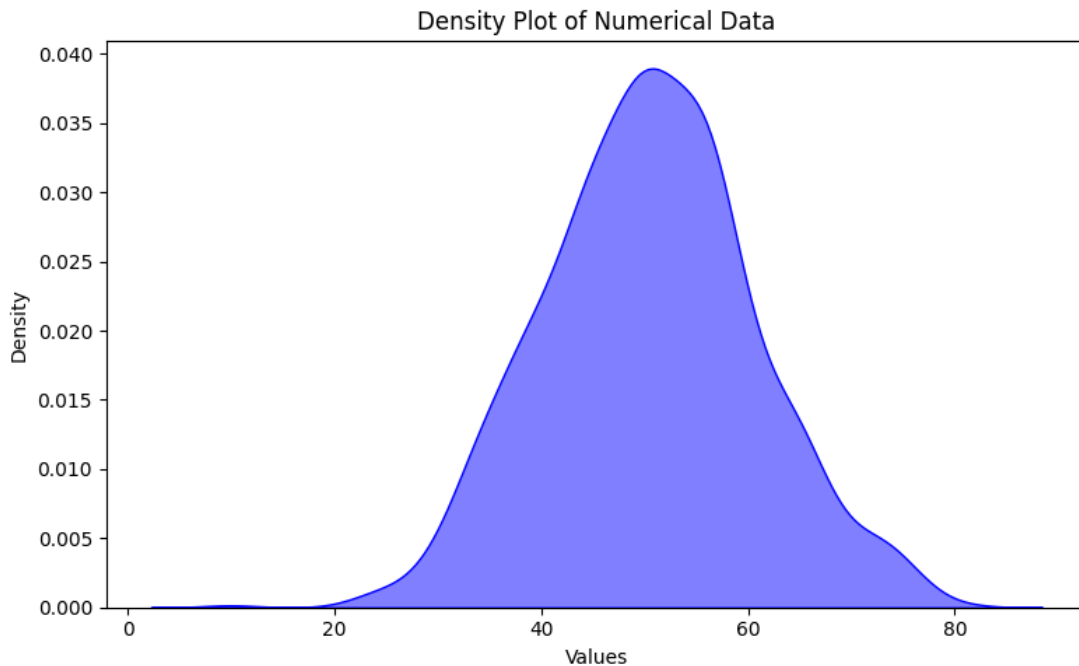
A histogram aims to approximate the underlying probability density function that generated the data by binning and counting observations. Kernel density estimation (KDE) presents a different solution to the same problem. Rather than using discrete bins, a KDE plot smooths the observations with a Gaussian kernel, producing a continuous density estimate:

Code Example:

```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
# Example numerical data
data = np.random.normal(loc=50, scale=10, size=1000) # Normal distribution

# KDE (Kernel Density Estimation) plot
plt.figure(figsize=(8, 5))
sns.kdeplot(data, color='blue', fill=True, alpha=0.5)
plt.title('Density Plot of Numerical Data')
plt.xlabel('Values')
plt.ylabel('Density')
plt.tight_layout()
plt.show()
```

Output:



Try:

1. Write a program to Generates a DataFrame with numerical data for five attributes (e.g., Attribute1, Attribute2, Attribute3, Attribute4, Attribute5).
2. Write a program to Plots the **distributions** (or **densities**) of the numerical attributes using a **kernel density estimate (KDE)** plot.
3. Write a program to Ensure that each attribute is plotted on the same graph for comparison, with different colors for each attribute, Include a title and appropriate labels for the plot.
4. Write a program to plot a density plot on given the dataset 'tips' and calculate what was the most common tip given by a customer.
5. Write a program to plot a density plot on given the dataset 'tips' and calculate what was the most common tip given by a customer using plot.kde() function.

4. Interpret the results .

Numerical Data (Histograms and Densities):

- **Histogram:**
 - Shows the frequency of data in bins.
 - Example: If the data has a single peak, it might indicate a normal distribution.
- **Density Plot:**
 - Highlights the smooth distribution of data.
 - Example: Multiple peaks might indicate bimodal or multimodal distributions.

Categorical Data (Bar Charts):

- Displays the counts of each category.
- Example: If one category has significantly higher counts, it might indicate a skew in the data distribution.

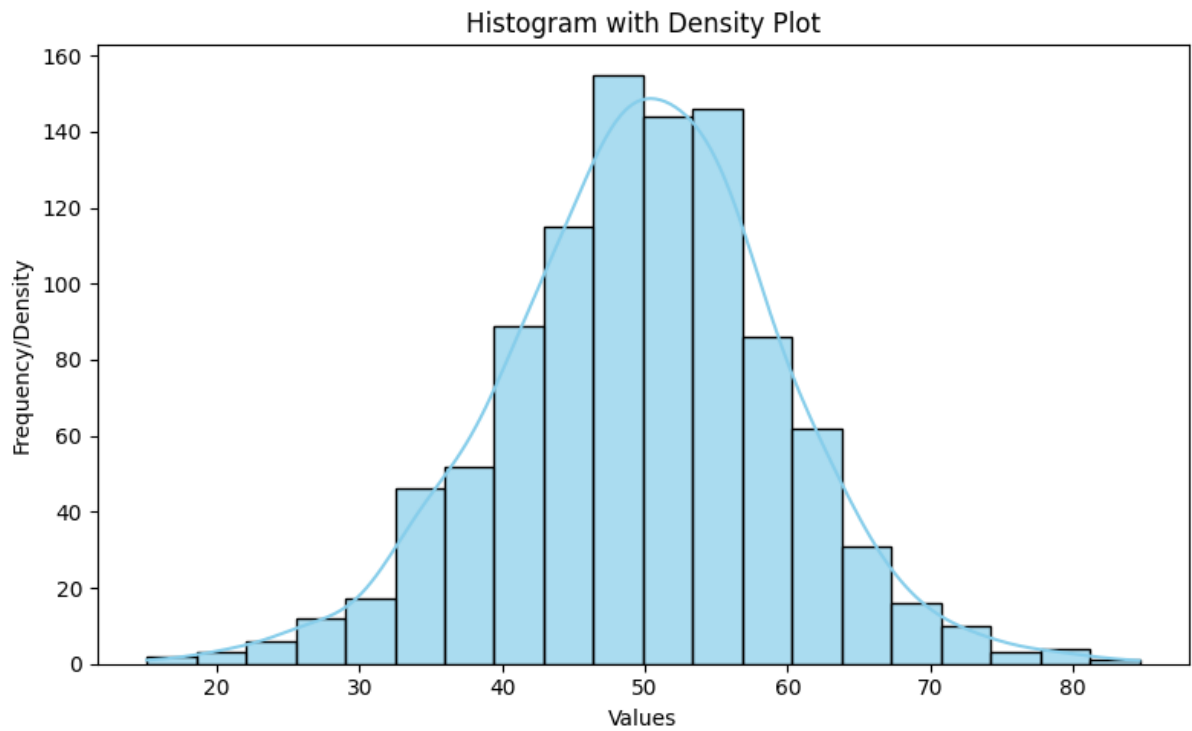
```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
# Example numerical data
data = np.random.normal(loc=50, scale=10, size=1000) # Normal distribution

plt.figure(figsize=(8, 5))
```



```
sns.histplot(data, bins=20, kde=True, color='skyblue', edgecolor='black', alpha=0.7)
plt.title('Histogram with Density Plot')
plt.xlabel('Values')
plt.ylabel('Frequency/Density')
plt.tight_layout()
plt.show()
```

Output:



10. Visualization using box-and-whisker plots.

Box-and-whisker plots, also called box plots, are effective for visualizing the distribution of numerical data through rank statistics. They summarize key aspects of the data, including the median, quartiles, and potential outliers.

1. Compute the rank statistics of numerical attributes.

Rank statistics are essential in various statistical analyses, especially when dealing with ordinal data or when the assumptions of parametric tests are not met. Here's how you can compute rank statistics for numerical attributes, along with an example using Python and pandas:

Code Example:

```
import numpy as np
import pandas as pd

# Example numerical data
data = {
    'Attribute1': np.random.normal(50, 10, 100), # Normally distributed
    'Attribute2': np.random.uniform(30, 70, 100), # Uniformly distributed
}

df = pd.DataFrame(data)

# Compute rank statistics
rank_stats = df.describe(percentiles=[0.25, 0.5, 0.75])
print(rank_stats)
```

Output:

	Attribute1	Attribute2
count	100	100
mean	49.41509	51.03182
std	9.717994	11.6534
min	26.28979	31.0754
25%	42.99788	40.33386
50%	49.84461	53.17149
75%	55.4665	60.88231
max	76.36262	69.90528

Try :

1. Write a program to Generates a DataFrame with numerical data for five attributes (e.g., Attribute1, Attribute2, Attribute3, Attribute4, Attribute5).
2. Write a program to Computes the **rank** of each value within each attribute.
3. Write a program to Computes the **rank statistics** (such as the **mean rank**, **maximum rank**, **minimum rank**, and **rank of a specific value**) for each attribute.
4. Write a program to Display the rank statistics for each attribute in a readable format.

2. Create the box-and-whisker plots of numerical attributes.

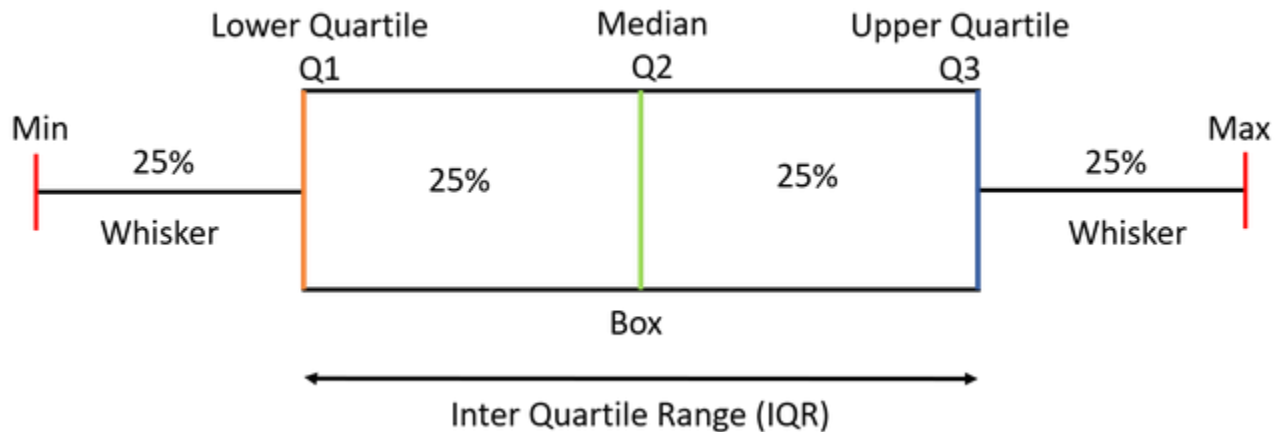
Box Plot is a graphical method to visualize data distribution for gaining insights and making informed decisions. Box plot is a type of chart that depicts a group of numerical data through their

quartiles.

Elements of Box Plot

A box plot gives a five-number summary of a set of data which is-

- **Minimum** – It is the minimum value in the dataset excluding the outliers.
- **First Quartile (Q1)** – 25% of the data lies below the First (lower) Quartile.
- **Median (Q2)** – It is the mid-point of the dataset. Half of the values lie below it and half above.
- **Third Quartile (Q3)** – 75% of the data lies below the Third (Upper) Quartile.
- **Maximum** – It is the maximum value in the dataset excluding the outliers.
- **IQR (Interquartile Range):** $IQR = Q3 - Q1$



```
import numpy as np
import pandas as pd

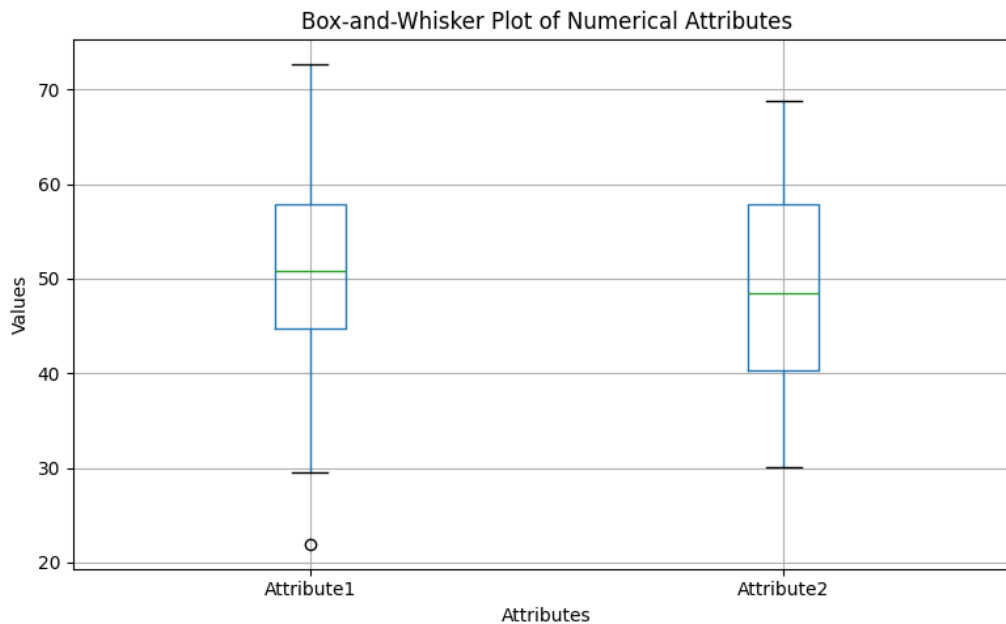
# Example numerical data
data = {
    'Attribute1': np.random.normal(50, 10, 100), # Normally distributed
    'Attribute2': np.random.uniform(30, 70, 100), # Uniformly distributed
}

df = pd.DataFrame(data)

import matplotlib.pyplot as plt

# Box plot
plt.figure(figsize=(8, 5))
df.boxplot()
plt.title('Box-and-Whisker Plot of Numerical Attributes')
plt.xlabel('Attributes')
plt.ylabel('Values')
plt.tight_layout()
plt.show()
```

Output:



Try :

- 1 Write a program to Generates a DataFrame with numerical data for five attributes (e.g., Attribute1, Attribute2, Attribute3, Attribute4, Attribute5).
- 2 Write a program to Creates **box-and-whisker plots** for each numerical attribute.
- 3 Write a program to Customize the plot with titles, axis labels, and grid lines.
- 4 Write a program to Display all box plots in a single figure.

3. Interpret the results .

Key Observations from Box Plots:

1. **Median:**
 - The line inside the box represents the median (Q2), showing the central tendency of the data.
 - Example: If the median is closer to the bottom of the box, the data is skewed towards higher values.
2. **IQR (Box Height):**
 - The height of the box indicates the interquartile range (Q3 - Q1).
 - Example: A taller box implies a wider spread of the middle 50% of the data.
3. **Whiskers:**
 - Extend from Q1 to the smallest value within $1.5 \times \text{IQR}$ and from Q3 to the largest value within $1.5 \times \text{IQR}$.
 - Example: Longer whiskers suggest data spread beyond the central range.
4. **Outliers:**
 - Points outside the whiskers are potential outliers.
 - Example: Outliers may indicate errors, rare events, or interesting deviations.
5. **Skewness:**
 - If the median is not centered in the box, the data is skewed.
 - Example: A left-skewed distribution has the median closer to Q3.

Customizing Box Plots

- You can customize the box plots for better analysis:

Grouped Box Plots:

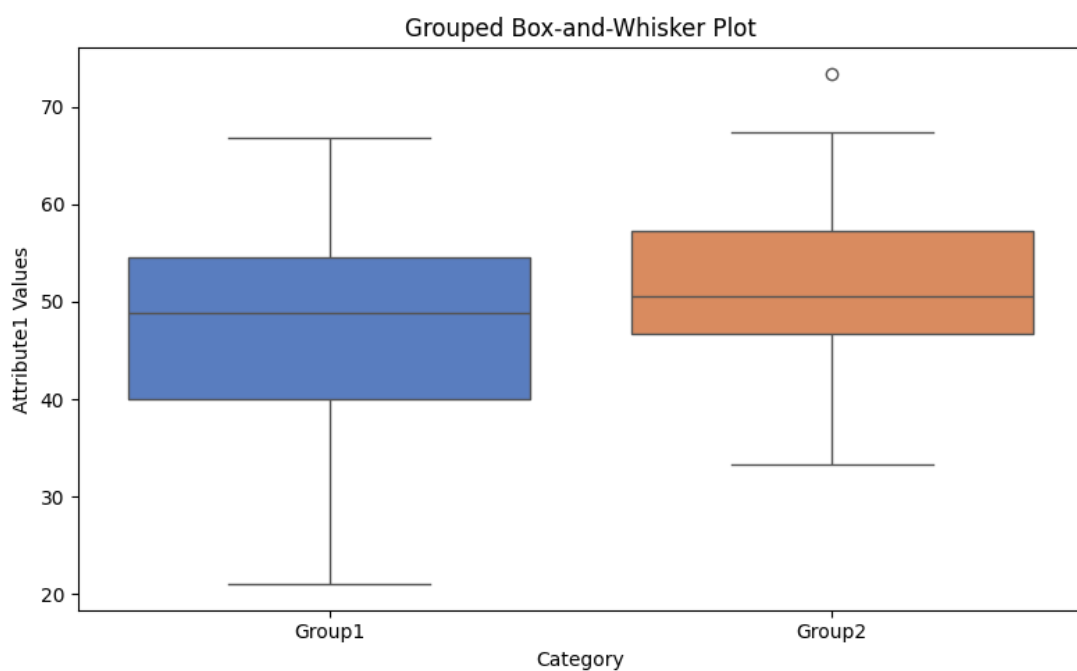
```
import numpy as np
import pandas as pd
import seaborn as sns
# Example numerical data
data = {
    'Attribute1': np.random.normal(50, 10, 100), # Normally distributed
    'Attribute2': np.random.uniform(30, 70, 100), # Uniformly distributed
}

df = pd.DataFrame(data)

import matplotlib.pyplot as plt
# Adding a categorical column for grouping
df['Category'] = np.random.choice(['Group1', 'Group2'], size=100)

# Grouped box plots
plt.figure(figsize=(8, 5))
sns.boxplot(data=df, x='Category', y='Attribute1', palette='muted')
plt.title('Grouped Box-and-Whisker Plot')
plt.xlabel('Category')
plt.ylabel('Attribute1 Values')
plt.tight_layout()
plt.show()
```

Output:



Horizontal Box Plots:

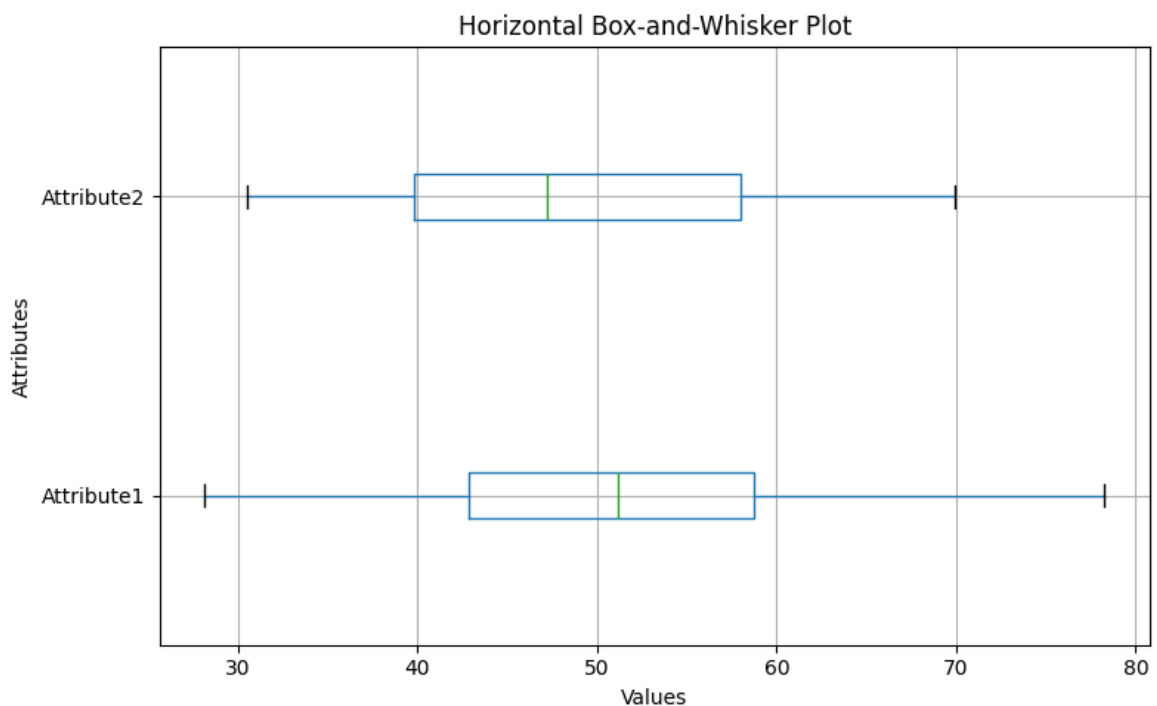
```
import numpy as np
import pandas as pd
import seaborn as sns
# Example numerical data
data = {
    'Attribute1': np.random.normal(50, 10, 100), # Normally distributed
    'Attribute2': np.random.uniform(30, 70, 100), # Uniformly distributed
}

df = pd.DataFrame(data)

import matplotlib.pyplot as plt
# Adding a categorical column for grouping
df['Category'] = np.random.choice(['Group1', 'Group2'], size=100)

# Grouped box plots
plt.figure(figsize=(8, 5))
df.boxplot.vert=False)
plt.title('Horizontal Box-and-Whisker Plot')
plt.xlabel('Values')
plt.ylabel('Attributes')
plt.tight_layout()
plt.show()
```

Output:



Try:

- 1 Given a numerical dataset, how do you interpret measures of central tendency (mean, median, mode) from the results

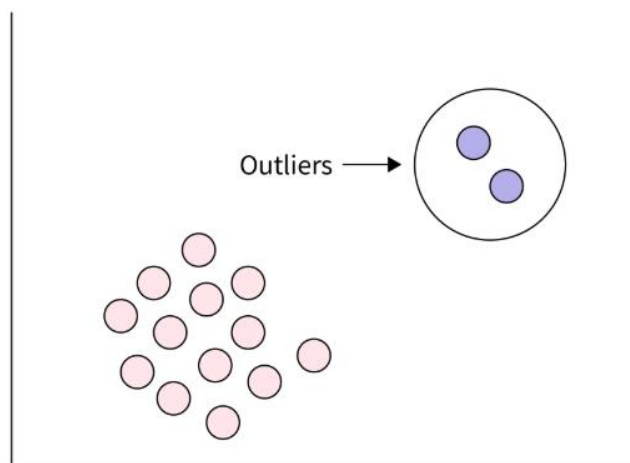
- 2 Given a dataset with significant uncertainty (e.g., missing data or noisy data), how do you interpret the results of your analysis?

11. Handling outliers in the data.

Outliers are the observations in a dataset that deviate significantly from the rest of the data. In any data science project, it is essential to identify and handle outliers, as they can have a significant impact on many statistical methods, such as means, standard deviations, etc., and the performance of ML models. Outliers can sometimes indicate errors or anomalies in the data.

1. Identify the outliers using quartile method.

- In statistics, any observations or data points that deviate significantly and do not conform with the rest of the observation or data points in a dataset are called outliers. Outliers are extreme values in a feature or dataset. For example, if you have a dataset with a feature height. The majority of the values in this feature range between 4.5–6.5 feet, but there is one value with 10 feet. This value would be considered an outlier, as it is not only an extreme value but an impossible height as well.
- Outliers are also called **aberrations**, **abnormal points**, **anomalies**, etc. It is essential to detect and handle outliers in a dataset as it can have a significant impact on many statistical methods, such as mean, variance, etc., and the performance of the ML models. It can lead to misleading, inconsistent, and inaccurate results if they are not properly accounted for.



The **quartile method** identifies outliers based on the interquartile range (IQR):

Steps:

1. Compute Q1 (25th percentile) and Q3 (75th percentile).
2. Calculate the IQR: $IQR = Q3 - Q1$.
3. Define lower and upper bounds:
4. Lower Bound = $Q1 - 1.5 \times IQR$
5. Upper Bound = $Q3 + 1.5 \times IQR$
6. Outliers are values outside these bounds.

Code Example:

```
import numpy as np
import pandas as pd

# Example numerical data
data = {
    'Attribute': np.concatenate([np.random.normal(50, 10, 100), [150, 170]]) # Adding
    outliers
```

```

}
df = pd.DataFrame(data)

# Quartile method
Q1 = df['Attribute'].quantile(0.25)
Q3 = df['Attribute'].quantile(0.75)
IQR = Q3 - Q1

lower_bound = Q1 - 1.5 * IQR
upper_bound = Q3 + 1.5 * IQR

# Identify outliers
outliers_quartile = df[(df['Attribute'] < lower_bound) | (df['Attribute'] > upper_bound)]
print("Outliers (Quartile Method):")
print(outliers_quartile)

```

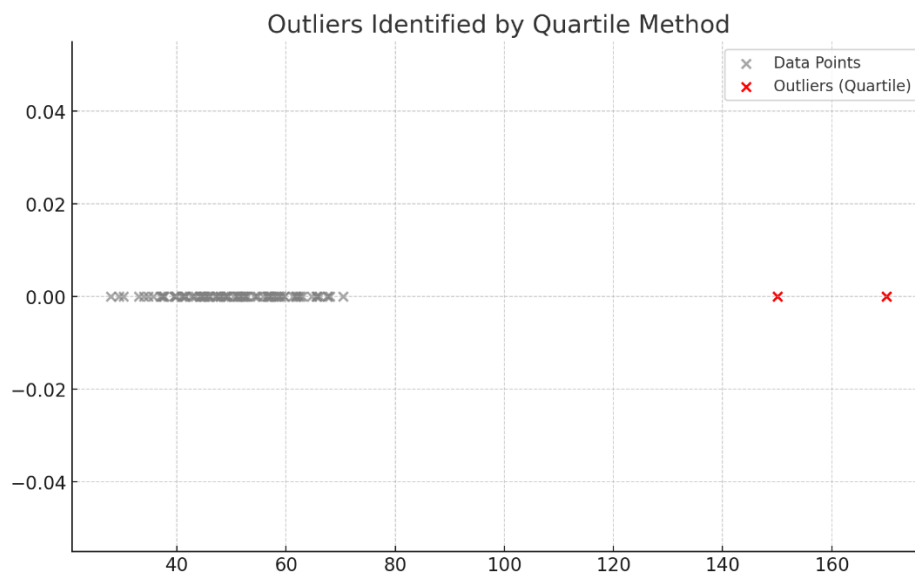
Output:

Outliers (Quartile Method):

```

Attribute
1    22.358335
100  150.000000
101  170.000000

```



Try:

1. Write a Python program to create a boxplot for a dataset and visually identify the outliers. Use the Titanic dataset to identify outliers in the age column.
2. Write a Python program to identify outliers in the daily closing prices of stocks using the IQR method. Use a dataset of historical stock prices for this analysis.
3. Write a Python program to remove outliers from a dataset using the quartile method. Use a dataset with numerical and categorical columns and ensure only numerical columns are processed

2. Identify the outliers using standard deviation method.

The **standard deviation method** identifies outliers based on how far values deviate from the mean:

Steps:

1. Compute the mean (μ) and standard deviation (σ).
2. Define the thresholds:
 - a. Lower Bound = $\mu - k \cdot \sigma$
 - b. Upper Bound = $\mu + k \cdot \sigma$
 - c. Common k values are 2 or 3.
3. Outliers are values outside these bounds.

Code Example:

```
import numpy as np
import pandas as pd

# Example numerical data
data = {
    'Attribute': np.concatenate([np.random.normal(50, 10, 100), [150, 170]]) # Adding outliers
}
df = pd.DataFrame(data)

mean = df['Attribute'].mean()
std = df['Attribute'].std()

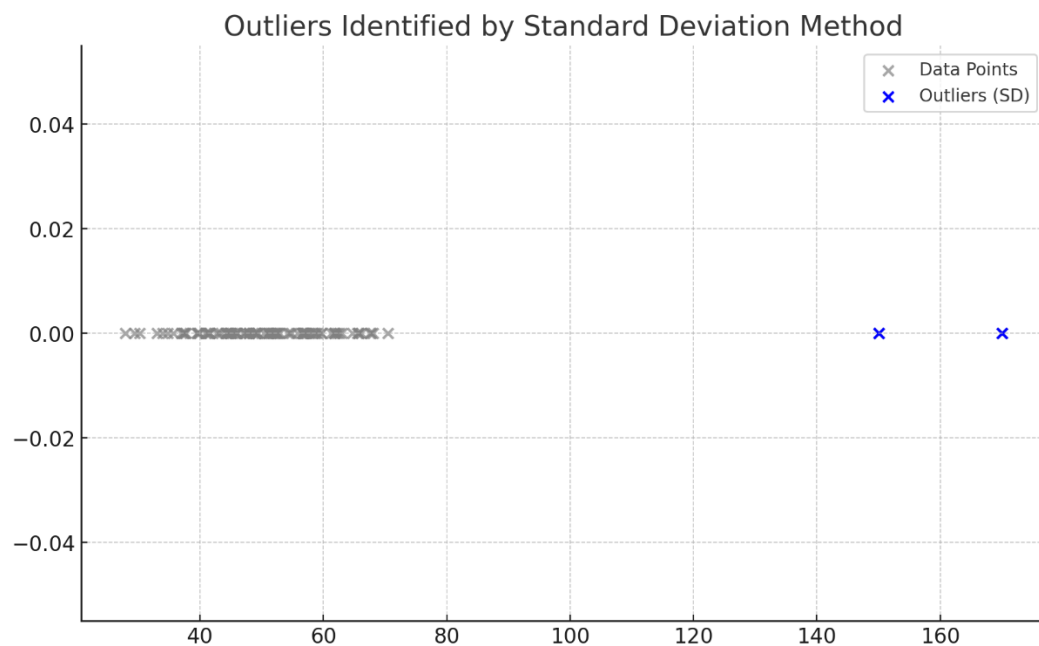
k = 3 # Using 3 standard deviations
lower_bound_sd = mean - k * std
upper_bound_sd = mean + k * std

# Identify outliers
outliers_sd = df[(df['Attribute'] < lower_bound_sd) | (df['Attribute'] > upper_bound_sd)]
print("Outliers (Standard Deviation Method):")
print(outliers_sd)
```

Output:

Outliers (Standard Deviation Method):

	Attribute
100	150.0
101	170.0



Try:

1. Write a Python program to calculate the mean and standard deviation of a dataset and identify outliers as data points more than 2 standard deviations away from the mean. Use a synthetic dataset for demonstration.
2. Write a Python program to calculate the mean and standard deviation of a dataset before and after removing outliers. Analyze the impact of outliers on these measures.
3. Write a Python program to create a data preprocessing pipeline that includes outlier detection using the standard deviation method. Apply this pipeline to a dataset with mixed data types.

3. Compare the performance of two methods.

Comparison:

1. **Quartile Method:**
 - Robust to skewed distributions.
 - May fail for datasets with highly irregular distributions.
2. **Standard Deviation Method:**
 - Assumes normality; less effective for skewed or non-normal data.
 - More sensitive to extreme values in highly skewed datasets.

Code to Compare:

```
import numpy as np
import pandas as pd

# Example numerical data
data = {
    'Attribute': np.concatenate([np.random.normal(50, 10, 100), [150, 170]]) # Adding outliers
}
df = pd.DataFrame(data)
# Quartile method
Q1 = df['Attribute'].quantile(0.25)
```

```

Q3 = df['Attribute'].quantile(0.75)
IQR = Q3 - Q1

lower_bound = Q1 - 1.5 * IQR
upper_bound = Q3 + 1.5 * IQR

mean = df['Attribute'].mean()
std = df['Attribute'].std()

k = 3

# Using 3 standard deviations
lower_bound_sd = mean - k * std
upper_bound_sd = mean + k * std

# Identify outliers
outliers_quartile = df[(df['Attribute'] < lower_bound) | (df['Attribute'] > upper_bound)]
print("Outliers (Quartile Method):")
print(outliers_quartile)

# Identify outliers
outliers_sd = df[(df['Attribute'] < lower_bound_sd) | (df['Attribute'] > upper_bound_sd)]
print("Outliers (Standard Deviation Method):")
print(outliers_sd)

# Number of outliers identified
print("Number of Outliers (Quartile Method):", len(outliers_quartile))
print("Number of Outliers (Standard Deviation Method):", len(outliers_sd))

```

Output:

Outliers (Quartile Method):

```

Attribute
0    86.167303
100  150.000000
101  170.000000

```

Outliers (Standard Deviation Method):

```

Attribute
100    150.0
101    170.0

```

Number of Outliers (Quartile Method): 3

Number of Outliers (Standard Deviation Method): 2

Try:

1. Write a Python program to compute and visualize the overlap between outliers detected by two methods. Use a Venn diagram to show the overlap.
2. Write a Python program to create side-by-side boxplots to visualize the results of two outlier detection methods. Compare the identified outliers visually.

4. Remove outliers from the data.

- This involves identifying and removing outliers from the dataset before training the model. Common methods include:
 - **Thresholding:** Outliers are identified as data points exceeding a certain threshold (e.g., Z-score > 3).
 - **Distance-based methods:** Outliers are identified based on their distance from their nearest neighbors.
 - **Clustering:** Outliers are identified as points not belonging to any cluster or belonging to very small clusters.

Remove outliers from the dataset using the chosen method.

Code Example:

```
# Removing outliers based on the Quartile Method
df_cleaned = df[(df['Attribute'] >= lower_bound) & (df['Attribute'] <= upper_bound)]
print("Data after Removing Outliers:")
print(df_cleaned.describe())
```

Output:

Outliers (Quartile Method):

Attribute

100 150.0

101 170.0

Data after Removing Outliers:

Attribute

count 100.000000

mean 49.821135

std 10.905372

min 27.573205

25% 42.710426

50% 49.927551

75% 57.236913

max 80.915658

Try:

1. Write a Python program to remove outliers from a dataset using the Z-score method. Use a sales dataset to remove products with unusually high or low prices.
2. Write a Python program to remove outliers from a streaming dataset (e.g., real-time sensor readings) using dynamic thresholds based on a rolling window.
3. Use a healthcare dataset to remove patients with abnormal values for metrics like blood pressure or cholesterol. Discuss the potential impact on downstream analyses.

5. Interpret the results

Before Handling Outliers:

- The dataset contains extreme values that may distort statistical analyses, such as mean and standard deviation.

After Removing Outliers:

- The dataset becomes more representative of the central trend.
- Statistical metrics like mean and standard deviation are less influenced by extreme values.

Summary

Aspect	Quartile Method	Standard Deviation Method
Assumptions	No assumptions on distribution	Assumes normal distribution
Robustness to Skewed Data	More robust	Less robust
Performance on Normal Data	Good	Very effective
Ease of Calculation	Moderate	Easy

Try:

1. Write a Python program to create boxplots or scatterplots before and after data transformation. How do you interpret changes in the visual representation of data after transformations like scaling or outlier removal.
2. After applying data preprocessing techniques like scaling, outlier removal, or normalization, how do you validate the results to ensure they are meaningful and accurate.

12. Working with Data Tables.

Data tables are powerful tools for organizing, analyzing, and visualizing data. They provide a structured way to represent information, making it easier to understand, manipulate, and extract insights.

Here's a breakdown of key aspects of working with data tables:

1. Creating Data Tables

- **Spreadsheet Software:** Tools like Excel, Google Sheets, and LibreOffice Calc offer built-in features for creating and managing data tables.
- **Programming Languages:**
 - **Python:** Libraries like pandas are widely used for creating, manipulating, and analyzing data tables (DataFrames).
 - **SQL:** Used for managing and querying data stored in relational databases.

2. Data Table Structure

- **Rows and Columns:** Data tables consist of rows and columns.
 - Rows represent individual data points or observations.
 - Columns represent specific attributes or variables.
- **Headers:** Column headers provide labels for the data in each column.

3. Data Types

- **Numerical:** Numbers (integers, floats)
- **Categorical:** Textual values representing categories (e.g., colors, countries)
- **Boolean:** True/False values
- **Date/Time:** Timestamps or dates

4. Key Operations

Data Entry: Manually entering data or importing data from external sources (CSV files, databases).

- **Data Cleaning:**
 - Handling missing values (imputation, removal)
 - Removing duplicates
 - Correcting errors
- **Data Transformation:**
 - Filtering data based on conditions
 - Sorting data by specific columns
 - Grouping data and calculating summary statistics (e.g., mean, median, sum)
 - Creating new columns based on existing ones (e.g., calculations, transformations)
- **Data Analysis:**
 - Performing statistical analyses (e.g., regression, hypothesis testing)
 - Creating visualizations (charts, graphs) to explore and understand data patterns.

1. Joining the data tables.

In pandas, joining data tables involves merging or concatenating two or more tables based on a common key or index. There are different types of joins, including inner, outer, left, and right joins.

Types of joins:

- **Inner Join:** Only includes matching rows from both tables.
- **Outer Join:** Includes all rows from both tables, filling in missing values with NaN.
- **Left Join:** Includes all rows from the left table and only matching rows from the right table.
- **Right Join:** Includes all rows from the right table and only matching rows from the left table.

Example:

```
import pandas as pd

# Create two data tables (dataframes)
df1 = pd.DataFrame({
    'ID': [1, 2, 3, 4],
    'Name': ['Alice', 'Bob', 'Charlie', 'David']
})

df2 = pd.DataFrame({
    'ID': [3, 4, 5, 6],
    'Age': [25, 30, 35, 40]
})

# Inner Join (only matching rows)
joined_df = pd.merge(df1, df2, on='ID', how='inner')
print(joined_df)
```

Output:

	ID	Name	Age
0	3	Charlie	25
1	4	David	30

Try:

1. Write a Python program to perform a self-join on a table. Use an example dataset of employees and their managers to demonstrate how to retrieve hierarchical relationships.
2. Write a Python program to join two large CSV files in chunks using pandas.
3. Write a Python program to join two time-series datasets based on their timestamps.

2. Exercises on contingency tables.

A **contingency table** (also known as a cross-tabulation) is used to display the frequency distribution of variables. It helps examine the relationship between two categorical variables.

A *contingency table* provides a way of portraying data that can facilitate calculating probabilities. The table helps in determining conditional probabilities quite easily. The table displays sample values in relation to two different variables that may be dependent or contingent on one another.

Example:

```
import pandas as pd

# Example data
data = {'Gender': ['Male', 'Female', 'Male', 'Female', 'Male'],
        'Purchased': ['Yes', 'No', 'Yes', 'Yes', 'No']}

df = pd.DataFrame(data)

# Create a contingency table (cross-tabulation)
contingency_table = pd.crosstab(df['Gender'], df['Purchased'])
print(contingency_table)
```

Output:

Purchased	No	Yes
Gender		
Female	1	1
Male	1	2

This creates a table that shows the frequency of each combination of gender and purchase status.

Try:

1. Write a Python program to create a contingency table from a dataset. Use the Titanic dataset to display the counts of survivors and non-survivors by gender.
2. Create a contingency table showing the relationship between two categorical variables and compute its marginal totals.
3. Write a Python program to perform a Chi-Square test of independence on a contingency table. Use a dataset to test whether gender and purchase decision are independent variables.

3. Exercises on grouping data.

In pandas, you can group data by one or more columns and perform operations like summing, averaging, or counting the grouped data.

Example:

```
import pandas as pd

# Sample data
data = {'Category': ['A', 'B', 'A', 'B', 'A', 'B'],
        'Value': [10, 20, 30, 40, 50, 60]}
```

```
df = pd.DataFrame(data)
```

```
# Group by 'Category' and calculate the sum of 'Value'  
grouped_df = df.groupby('Category')['Value'].sum()  
print(grouped_df)
```

Output:

Category

A 90

B 120

Name: Value, dtype: int64

Try:

1. Write a Python program to group data by a single column and compute the mean of another column. Use the Titanic dataset to calculate the average age of passengers grouped by their class.
2. Write a Python program to group data by multiple columns. Use a dataset to find the total revenue for each combination of product category and region.
3. Write a Python program to rank items within each group. Use a dataset to rank employees by their sales performance within each department

13. Data Scaling and Transformation.

Data scaling and transformation are essential preprocessing techniques in machine learning to ensure that your data is in a suitable format for analysis and modeling. These methods address issues like varying scales, skewed distributions, and outliers, which can significantly impact the performance of your machine learning models.

A. Scaling the data using different Python scalers.

Step 1: Import necessary libraries:

- pandas for data manipulation.
- StandardScaler, MinMaxScaler, RobustScaler from sklearn.preprocessing for different scaling methods.

Step 2: Create sample data:

- Create a sample DataFrame with two columns: 'Age' and 'Income'.

Step 3: StandardScaler:

- Creates a StandardScaler object.
- fit_transform() calculates the mean and standard deviation of the data and transforms the data to have zero mean and unit variance.

```
import pandas as pd  
from sklearn.preprocessing import StandardScaler, MinMaxScaler, RobustScaler  
# Sample data (replace with your actual data)  
data = {'Age': [25, 30, 45, 22, 18],  
        'Income': [50000, 70000, 120000, 45000, 30000]}  
df = pd.DataFrame(data)  
# 1. StandardScaler (Standardization)  
scaler = StandardScaler()  
df_standardized = df.copy()  
df_standardized[['Age', 'Income']] = scaler.fit_transform(df[['Age', 'Income']])  
# 2. MinMaxScaler (Normalization)  
scaler = MinMaxScaler()  
df_normalized = df.copy()  
df_normalized[['Age', 'Income']] = scaler.fit_transform(df[['Age', 'Income']])
```



```
# 3. RobustScaler (Robust to outliers)
scaler = RobustScaler()
df_robust = df.copy()
df_robust[['Age', 'Income']] = scaler.fit_transform(df[['Age', 'Income']])
# Print scaled dataframes
print("Standardized Data:\n", df_standardized)
print("\nNormalized Data:\n", df_normalized)
print("\nRobust Scaled Data:\n", df_robust)
```

2. **MinMaxScaler:**

- Creates a MinMaxScaler object.
- `fit_transform()` scales the data to a specific range (usually 0 to 1).

3. **RobustScaler:**

- Creates a RobustScaler object.
- `fit_transform()` is less sensitive to outliers compared to StandardScaler. It uses the median and interquartile range for scaling.

4. **Print scaled dataframes:**

- Prints the original and scaled dataframes for each scaling method.

Key Points:

Step 4: Choose the appropriate scaler:

- **StandardScaler:** Suitable for many cases, especially when the data is normally distributed.
- **MinMaxScaler:** Useful when you need to scale data to a specific range (e.g., for neural networks).
- **RobustScaler:** More robust to outliers than StandardScaler.

Apply scaling to relevant features:

- Typically, you would scale only the numerical features in your dataset.

Fit and transform:

- `fit_transform()` calculates the scaling parameters (e.g., mean, standard deviation) from the training data and applies the transformation.
- Use `fit_transform()` on the training data and `transform()` on the test data to ensure consistency.

Try:

1. Write a Python program to demonstrate how to scale a dataset using the MinMaxScaler from the `sklearn.preprocessing` module.
2. Write a Python program to compare the effects of different scalers, including StandardScaler, MinMaxScaler, MaxAbsScaler, and RobustScaler, on a synthetic dataset with outliers. Visualize the scaled results using box plots.

b. Normalization as a special case of data scaling.

Step 1: Import necessary libraries:

- `pandas` for data manipulation.
- `MinMaxScaler` from `sklearn.preprocessing` for normalization.

Step 2: Create sample data:

- Create a sample DataFrame with two columns: 'Age' and 'Income'.

Step 3 Create a MinMaxScaler object:

- `MinMaxScaler()` creates an object that will scale the data to a specific range (default: 0 to 1).

Step 4: Fit and transform the data:

- `scaler.fit_transform(df[['Age', 'Income']])` calculates the minimum and maximum values of the 'Age' and 'Income' columns and then scales the data to the range 0 to 1 using the following formula:
- $$X_{\text{scaled}} = (X - X_{\text{min}}) / (X_{\text{max}} - X_{\text{min}})$$
- The scaled values are then assigned back to the corresponding columns in the `df_normalized` DataFrame.

Step 5: Print the normalized data:

- Print the resulting DataFrame with the normalized values.

Step 6: Sample Code

```
import pandas as pd
from sklearn.preprocessing import StandardScaler, MinMaxScaler, RobustScaler
# Sample data (replace with your actual data)
data = {'Age': [25, 30, 45, 22, 18],
        'Income': [50000, 70000, 120000, 45000, 30000]}
df = pd.DataFrame(data)
# 1. StandardScaler (Standardization)
scaler = StandardScaler()
df_standardized = df.copy()
df_standardized[['Age', 'Income']] = scaler.fit_transform(df[['Age', 'Income']])
# 2. MinMaxScaler (Normalization)
scaler = MinMaxScaler()
df_normalized = df.copy()
df_normalized[['Age', 'Income']] = scaler.fit_transform(df[['Age', 'Income']])
# 3. RobustScaler (Robust to outliers)
scaler = RobustScaler()
df_robust = df.copy()
df_robust[['Age', 'Income']] = scaler.fit_transform(df[['Age', 'Income']])
# Print scaled dataframes
print("Standardized Data:\n", df_standardized)
print("\nNormalized Data:\n", df_normalized)
print("\nRobust Scaled Data:\n", df_robust)
```

Note:

- Normalization scales the data to a specific range (typically 0 to 1), making all features have the same scale.
- MinMaxScaler is a common technique for normalization in machine learning.
- Normalization is useful when:
 - You want to ensure all features have the same influence on the model.
 - You are using algorithms that are sensitive to feature scaling (e.g., some neural network algorithms).

Try:

1. What is normalization, and how does it differ from other data scaling techniques? Write a Python program to normalize a dataset using the MinMaxScaler and demonstrate how the transformed data lies within the range [0, 1].

2. Write a Python program to demonstrate the impact of normalization on the performance of a KNN classifier using the Iris dataset.
3. Write a Python program to normalize the MNIST dataset's pixel values to the range [0, 1] and train a simple neural network using TensorFlow or PyTorch

C. Data transformation using standardization.

Step 1: import necessary libraries:

- pandas for data manipulation.
- StandardScaler from sklearn.preprocessing for standardization.

Step 2: Create sample data:

- Create a sample DataFrame with two columns: 'Age' and 'Income'.

Step 3: Create a StandardScaler object:

- StandardScaler() creates an object that will standardize the data.

Step 4: Fit and transform the data:

- scaler.fit_transform(df[['Age', 'Income']]) calculates the mean and standard deviation of the 'Age' and 'Income' columns and then standardizes the data using the following formula:
- $z = (x - \text{mean}) / \text{standard_deviation}$
- The standardized values (z-scores) have a mean of 0 and a standard deviation of 1.
- The scaled values are then assigned back to the corresponding columns in the df_standardized DataFrame.

Step 5: Print the standardized data:

- Print the resulting DataFrame with the standardized values.

```
import pandas as pd
from sklearn.preprocessing import StandardScaler
# Sample data
data = {'Age': [25, 30, 45, 22, 18],
        'Income': [50000, 70000, 120000, 45000, 30000]}
df = pd.DataFrame(data)
# Create a StandardScaler object
scaler = StandardScaler()
# Fit and transform the data
df_standardized = df.copy()
df_standardized[['Age', 'Income']] = scaler.fit_transform(df[['Age', 'Income']])
# Print the standardized data
print("Standardized Data:\n", df_standardized)
```

Note:

- **Standardization** transforms the data to have zero mean and unit variance, making it easier for machine learning algorithms to work with.
- It's particularly useful when features have different scales or when algorithms are sensitive to feature scaling.
- Standardization is often used in conjunction with algorithms like Support Vector Machines (SVM) and linear regression.

Try:

1. Write a program to standardize a dataset manually using the formula $Z = \frac{X - \mu}{\sigma}$

2. Write a function to standardize a dataset manually without using external libraries. Apply the function to a synthetic dataset and verify its correctness by comparing it to the StandardScaler from sklearn.preprocessing.

D. Compare the results and interpret.

Step 1: Import necessary libraries:

pandas for data manipulation.

StandardScaler, MinMaxScaler, RobustScaler from sklearn.preprocessing for different scaling methods.

Step 2: Create sample data:

Create a sample DataFrame with two columns: 'Age' and 'Income'.

Step 3: Create scaler objects:

Create instances of StandardScaler, MinMaxScaler, and RobustScaler.

Step 4: Scale the data:

Apply fit_transform() to each scaler to scale the data.

Step 5: Compare and interpret results:

Print the original and scaled DataFrames.

Calculate and print summary statistics (mean, standard deviation, min, max, quartiles) for each DataFrame.

```
import pandas as pd
from sklearn.preprocessing import StandardScaler, MinMaxScaler, RobustScaler
# Sample data
data = {'Age': [25, 30, 45, 22, 18],
        'Income': [50000, 70000, 120000, 45000, 30000]}
df = pd.DataFrame(data)
# Create scaler objects
standard_scaler = StandardScaler()
min_max_scaler = MinMaxScaler()
robust_scaler = RobustScaler()
# Scale the data
df_standardized = df.copy()
df_standardized[['Age', 'Income']] = standard_scaler.fit_transform(df[['Age', 'Income']])
df_normalized = df.copy()
df_normalized[['Age', 'Income']] = min_max_scaler.fit_transform(df[['Age', 'Income']])
df_robust = df.copy()
df_robust[['Age', 'Income']] = robust_scaler.fit_transform(df[['Age', 'Income']])
# Compare and interpret results
print("Original Data:\n", df)
print("\nStandardized Data:\n", df_standardized)
print("\nNormalized Data:\n", df_normalized)
print("\nRobust Scaled Data:\n", df_robust)
# Calculate and print summary statistics
print("\nSummary Statistics:")
print("Original Data:\n", df.describe())
print("\nStandardized Data:\n", df_standardized.describe())
print("\nNormalized Data:\n", df_normalized.describe())
print("\nRobust Scaled Data:\n", df_robust.describe())
```

Step 5: Interpretation:

Standardized Data:

Mean is close to 0.

Standard deviation is 1.

Data is centered around 0, making it suitable for algorithms that assume zero mean.

Step 6: Normalized Data:

Values are scaled between 0 and 1.

Useful for algorithms that require input features to be within a specific range.

Step 6: Robust Scaled Data:

Less sensitive to outliers compared to StandardScaler.

Uses median and interquartile range for scaling.

Try:

1. Train a Support Vector Machine (SVM) classifier on the Iris dataset without standardizing the features. Then, standardize the dataset using StandardScaler and train the classifier again. Compare the accuracy scores and interpret the results.
2. Write a custom function to standardize a dataset and compare the results with StandardScaler from sklearn.preprocessing. Interpret any differences and discuss the implications of using custom scaling methods.

14. Web Scrapping.

Web scraping, also known as web harvesting or web data extraction, is the process of automatically collecting and extracting data from websites. It involves using software or scripts to access the HTML code of a website and extract the desired information.

a. Scraping a list of items from a website.

Python example demonstrating how to scrape a list of items from a website, along with explanations:

Step 1: Import necessary libraries:

```
import requests
from bs4 import BeautifulSoup
```

- requests: This library allows you to fetch the HTML content of a webpage.
- BeautifulSoup: This library helps you parse the HTML content and extract specific data.

Step 2: Fetch the webpage content:

```
url = "https://www.example.com" # Replace with the actual URL
response = requests.get(url)
response.raise_for_status() # Raise an exception for bad status codes
soup = BeautifulSoup(response.content, "html.parser")
```

- Replace "https://www.example.com" with the URL of the website you want to scrape.
- requests.get(url) fetches the HTML content of the webpage.
- response.raise_for_status() checks if the request was successful (status code 200).
- BeautifulSoup(response.content, "html.parser") parses the HTML content using the html.parser.

Step 3: Find the elements containing the desired data:

```
items = soup.find_all("div", class_="item-container") # Replace with the appropriate HTML tags and attributes
```

- soup.find_all("div", class_="item-container") finds all <div> tags with the class "item-container" in the HTML. You need to inspect the HTML source of the webpage to determine the correct tags and attributes for finding the items you want to scrape.

Step 4: Extract the desired information from each item:

```
for item in items:
    name = item.find("h3", class_="item-name").text.strip()
```

```

price = item.find("span", class_="item-price").text.strip()
# Extract other relevant information (e.g., description, image URL)
print(f"Name: {name}")
print(f"Price: {price}")
# Print other extracted information
print("-" * 20)

```

- This code iterates through each item found in the previous step.
- `item.find("h3", class_="item-name").text.strip()` finds the `<h3>` tag with the class "item-name" within each item and extracts its text content, removing any leading/trailing whitespace.
- Similarly, `item.find("span", class_="item-price").text.strip()` extracts the price.
- You can adjust the code to extract other relevant information from each item by finding the corresponding HTML tags and attributes.
- The code then prints the extracted information.

Total program

```

import requests
from bs4 import BeautifulSoup
url = "https://www.example.com" # Replace with the actual URL
response = requests.get(url)
response.raise_for_status()
soup = BeautifulSoup(response.content, "html.parser")
items = soup.find_all("div", class_="item-container")
for item in items:
    name = item.find("h3", class_="item-name").text.strip()
    price = item.find("span", class_="item-price").text.strip()
    print(f"Name: {name}")
    print(f"Price: {price}")
    print("-" * 20)

```

Note:

- This is a basic example. You may need to adapt it based on the specific structure of the website you're scraping.
- Always check the website's terms of service before scraping. Some websites may prohibit or restrict scraping.
- Consider using a library like `scrapy` for more advanced web scraping tasks, which provides features like data pipelines, handling JavaScript, and more.

This example provides a foundation for scraping a list of items from a website. Remember to inspect the HTML source of the target website carefully to identify the correct HTML elements and attributes for extracting the desired data.

Try:

1. Write a Python program to scrape a list of product names, prices, and URLs from an e-commerce website. https://www.amazon.in/?&ext_vrnc=hi&tag=googhydrabk1-21&ref=pd_sl_7hz2t19t5c_e&adgrpid=58355126069&hvpone=&hvptwo=&hvadid=610644601173&hvpos=&hvnetw=g&hvrnd=2271425446877080510&hvqmt=e&hvdev=c&hvdvcmdl=&hvl ocint=&hvl ocphy= 9062186&hvtargid=kwd-10573980&hydadcr=14453_2316415.
2. Write a program to scrape a List of Job Openings from a Job Search Website. https://www.naukri.com/engineering-jobs?src=discovery_trendingWdgt_homepage_srch. Page | 110
3. Write a program to scrape a List of Books from an Online Bookstore

<https://www.bookswagon.com/>.

b. Scraping data from a table.

Step 1: Import necessary libraries:

- requests: To fetch the HTML content from the URL.
- BeautifulSoup: To parse the HTML and extract the table data.

Step 2: Define the scrape_table_data function:

- This function takes the URL of the webpage as input.
- It fetches the HTML content using requests.get(url).
- It checks for successful response using response.raise_for_status().
- It parses the HTML content using BeautifulSoup.
- It finds the first <table> tag on the page using soup.find('table').
- If a table is found:
- It extracts all table rows (<tr>) using table.find_all('tr').
- It iterates through each row:
- Extracts all table cells (<td>) within the row using row.find_all('td').
- Extracts the text content of each cell, strips whitespace, and stores it in a list.
- Appends the list of cell data to the data list.
- Returns the data list containing all rows of the table.
- If no table is found, it prints an error message and returns None.
- Includes error handling for potential requests.exceptions.RequestException.

Step 3: Usage of code:

- Sets the url to the actual URL of the webpage containing the table.
- Calls the scrape_table_data() function to get the table data.
- If data is successfully extracted, it iterates through each row and prints it.

Key Points:

- **HTML Structure:** This code assumes a basic HTML table structure with rows (<tr>) and cells (<td>). Adjust the code if the table structure is different.
- **Error Handling:** Includes basic error handling for network issues or if the table is not found on the page.
- **Flexibility:** You can modify the code to extract data from specific columns, handle different table structures, or handle more complex scenarios.

```
import requests
from bs4 import BeautifulSoup
def scrape_table_data(url):
    """
    Scrapes data from an HTML table given the URL.
    Args:
    url: The URL of the webpage containing the table.
```

```

Returns:
A list of lists, where each inner list represents a row of data.
"""
try:
    response = requests.get(url)
    response.raise_for_status() # Raise an exception for bad status codes
    soup = BeautifulSoup(response.content, "html.parser")
    table = soup.find('table') # Find the first table on the page
    if table:
        rows = table.find_all('tr')
        data = []
        for row in rows:
            cols = row.find_all('td') # Extract data from table cells (<td>)
            row_data = [col.text.strip() for col in cols]
            data.append(row_data)
        return data
    else:
        print("No table found on the page.")
        return None
except requests.exceptions.RequestException as e:
    print(f"Error fetching URL: {e}")
    return None
# Example usage:
url = "https://example.com/table_page.html" # Replace with the actual URL
table_data = scrape_table_data(url)
if table_data:
    for row in table_data:
        print(row)

```

To use this code:

- Replace "https://example.com/table_page.html" with the actual URL of the webpage you want to scrape.
- Run the Python script.
- This will print the extracted table data to the console. You can then further process this data as needed (e.g., save it to a file, perform calculations, etc.).

Try:

1. Write a program to scrape data from a table on Doctors Without Borders (Médecins Sans Frontières - MSF) www.msf.org website.
2. Write a program to scrape a Table with Headers on the above Doctors Without Borders website.
3. Write a program to scrape a Table with Pagination on an above Doctors Without Borders website.

C. Scraping images from a website.

The step-by-step guide to scraping images from a website using Python:

Step 1. Import necessary libraries:

```

import requests
from bs4 import BeautifulSoup
import os

```

- requests: To fetch the HTML content from the URL.

- BeautifulSoup: To parse the HTML and extract image URLs.
- os: To create filenames and handle file paths.

Step 2. Define the `scrape_images` function:

```
def scrape_images(url, save_dir="images"):
    """
    Scrapes images from a given URL and saves them to a specified directory
    Args:
        url: The URL of the webpage to scrape.
        save_dir: The directory to save the downloaded images (default: "images").
    """
    try:
        response = requests.get(url)
        response.raise_for_status() # Raise an exception for bad status codes
        soup = BeautifulSoup(response.content, "html.parser")

        images = soup.find_all("img")

        if not os.path.exists(save_dir):
            os.makedirs(save_dir)

        for i, image in enumerate(images):
            try:
                img_url = image["src"]
                img_data = requests.get(img_url).content
                img_name = f"image_{i}.jpg" # Customize filename as needed
                img_path = os.path.join(save_dir, img_name)
                with open(img_path, "wb") as handler:
                    handler.write(img_data)
                print(f"Downloaded {img_name} to {save_dir}")
            except Exception as e:
                print(f"Error downloading image: {e}")

    except requests.exceptions.RequestException as e:
        print(f"Error fetching URL: {e}")
```

- This function takes the URL and an optional `save_dir` as input.
- Fetches the HTML content using `requests.get(url)`.
- Parses the HTML content using BeautifulSoup.
- Finds all `` tags on the page using `soup.find_all("img")`.
- Creates the `save_dir` if it doesn't exist.
- Iterates through each image:
- Extracts the `src` attribute (image URL) using `image["src"]`.
- Fetches the image data using `requests.get(img_url).content`.
- Creates a filename for the image (e.g., `image_{i}.jpg`).
- Creates the full path to the image file.

- Saves the image data to the file.
- Prints a success message.
- Includes error handling for potential exceptions during image download.
- Includes error handling for potential exceptions during URL fetching.

Step 3: Usage of code

```
url = "https://www.example.com"
# Replace with the actual URL
scrape_images(url)
```

- Sets the url to the actual URL of the webpage containing the images.
- Calls the scrape_images() function to start scraping.

Notes:

- **HTML Structure:** This code assumes the image URLs are stored in the src attribute of the tag. Adjust the code if the HTML structure is different.
- **Error Handling:** Includes basic error handling for network issues, image download failures, and invalid image URLs.
- **Filename Customization:** Customize the filename generation logic as needed.
- **Image Types:** This code assumes JPG format. Modify for other formats.
- **Directory Creation:** Creates the save_dir if it doesn't exist.
- **Website Terms:** Always check website terms and robots.txt.
- **Dynamic Loading:** If images load dynamically, use Selenium or similar tools.

To use this code:

1. Replace "https://www.example.com" with the actual URL.
2. Run the Python script.

This will download the images to the specified directory (or "images" by default).

Try:

1. Write a program to scrape Images from a Gallery on the Doctors Without Borders www.msf.org website.
2. Write a program to scrape Images from a Search Results Page as www.msf.org website.

d. Scraping data with pagination.

The step-by-step guide on scraping data with pagination in Python, along with an example code:

Step 1: Import necessary libraries:

```
import requests
from bs4 import BeautifulSoup
```

- requests: Fetches HTML content from URLs.
- BeautifulSoup: Parses HTML content to extract data.

Step 2: Identify Pagination Mechanism:

- Inspect the website's HTML code to understand how pagination works.

- Look for patterns in URLs or HTML elements that change with different pages.

Step 3: Define the `scrape_page` function:

```
def scrape_page(url):
    """
    Scrapes data from a single page of a website.
    Args:
        url: The URL of the page to scrape.
    Returns:
        A list of extracted data (e.g., dictionaries, lists) or None if no data found.
    """
    try:
        response = requests.get(url)
        response.raise_for_status()
        soup = BeautifulSoup(response.content, "html.parser")
        # Extract data from the current page (replace with your specific logic)
        data = []
        # ... (your data extraction logic)
        return data
    except requests.exceptions.RequestException as e:
        print(f"Error fetching URL: {e}")
        return None
```

- This function takes a url as input.
- Fetches the HTML content using `requests.get(url)`.
- Parses the HTML content using `BeautifulSoup`.
- Replace the `# ... (your data extraction logic)` comment with your code to extract relevant data from the page.
- Returns the extracted data (`data`) or `None` if an error occurs or no data is found.

Step 4. Define the `scrape_all_pages` function:

```
def scrape_all_pages(base_url, pagination_param="page", start_page=1, end_page=None):
    """
    Scrapes data from all pages of a website using pagination.
    Args:
        base_url: The base URL of the pagination links (e.g., "https://example.com/products?").
        pagination_param: The query parameter used for pagination (e.g., "page").
        start_page: The starting page number (default: 1).
        end_page: The ending page number (default: None, scrape all pages).
    Returns:
        A list of all extracted data from all pages.
    """
    all_data = []
    for page_num in range(start_page, end_page + 1 if end_page else 1000): # Adjust max pages
        url = f"{base_url}{pagination_param}={page_num}"
        page_data = scrape_page(url)
        if page_data:
            all_data.extend(page_data) # Add data from each page
        else:
            break # Stop if no data found on a page (potential end of pagination)
```

return all_data

- This function takes the base_url, pagination_param, start_page, and end_page as input.
- Iterates through a range of page numbers (default: 1 to 1000, adjust as needed).
- Constructs the URL for each page using the base_url and pagination_param.
- Calls scrape_page(url) to extract data from each page.
- Appends the extracted data from each page to the all_data list.
- Stops iterating if no data is found on a page (indicating the end of pagination).
- Returns the list of all extracted data from all pages.

5. Usage of code:

```
base_url = "https://www.example.com/products?" # Replace with actual base URL
pagination_param = "page" # Replace if pagination uses a different parameter
start_page = 1 # Optional, start from a specific page
end_page = 5 # Optional, scrape only up to a certain page

all_data = scrape_all_pages(base_url, pagination_param, start_page, end_page)

if all_data:
    # Process the scraped data (e.g., print, save to file, etc.)
    for item in all_data:
        print(item) # Example: Print each item
else:
    print("No data found")
```

Try:

- 1 Write a program to scrape data with pagination from a website involves navigating through multiple pages to collect all the desired information.
- 2 Write a program to scrape Images from a Search Results Page as www.msf.org website.

V. TEXTBOOKS

1. R. Nageswara Rao, "Core Python Programming, 3ed: Covers fundamentals to advanced topics like OOPS, Exceptions, Data structures, Files, Threads, Net", Dreamtech press, 3rd edition, 2021.
2. Eric Jacqueline Kazil & Katharine Jarmul," Data Wrangling with Python", O'Reilly Media, Inc, 2016.

VI. REFERENCE BOOKS:

1. Dr. Tirthajyoti Sarkar, Shubhadeep," Data Wrangling with Python: Creating actionable data from raw sources", Packet Publishing Ltd, 2019.
2. Making sense of Data: A practical Guide to Exploratory Data Analysis and Data Mining, by Glenn J. Myatt.
3. Wes McKinney, "Python for Data Analysis: Data Wrangling with Pandas, NumPy, and IPython", O'Reilly, 2nd Edition, 2018.
4. Dr. John P. Hoffmann, "Principles of Data Management and Presentation", 1st edition, 2017.
5. Jake VanderPlas, "Python Data Science Handbook: Essential Tools for Working with Data", O'Reilly, 2017.
6. Y. Daniel Liang, "Introduction to Programming using Python", Pearson, 2012.

VIII. ELECTRONIC RESOURCES

1. <https://www.dataquest.io/blog/sci-kit-learn-tutorial/>
2. https://www.ibm.com/support/knowledgecenter/en/SS3RA7_sub/modeler_tutorial_ddita/modeler_tutorial_ddita-gentopic1.html
3. <https://archive.ics.uci.edu/ml/datasets.php>
4. <https://www.edx.org/course/analyzing-data-with-python>
5. [http://math.ecnu.edu.cn/~lfzhou/seminar/\[Joel_Grus\]_Data_Science_from_Scratch_First_Princ.pdf](http://math.ecnu.edu.cn/~lfzhou/seminar/[Joel_Grus]_Data_Science_from_Scratch_First_Princ.pdf)
6. <https://www.programmer-books.com/introducing-data-science-pdf/>

VIII. MATERIALS ONLINE

1. Course template
2. Lab Manual