**Section:4**

1. import numpy as np

random\_array = np.random.randint(1, 101, 20)

print(random\_array)

**Output:** [ 90 58 100 24 76 82 85 42 79 29 6 96 23 91 73 50 8 51 35]

2. import pandas as pd

temperatures = pd.Series([10, 15, 20, 25], index=["Monday", "Tuesday", "Wednesday", "Thursday"])

print(temperatures)

**Output:** Monday 10

Tuesday 15

Wednesday 20

Thursday 25

dtype: int64

3. mean\_temp = temperatures.mean()

std\_dev\_temp = temperatures.std()

size\_temp = temperatures.size

# Print the results

print("Mean temperature:", mean\_temp)

print("Standard deviation of temperature:", std\_dev\_temp)

print("Size of the temperature series:", size\_temp)

**Output:** Mean temperature: 17.5

Standard deviation of temperature: 6.454972243679028

Size of the temperature series: 4

6. import pandas as pd

# Create DataFrames

data\_X = {'Roll No': [10, 20, 30, 40], 'name': ['a', 'b', 'c', 'd']}

data\_Y = {'Roll No': [10, 20, 90, 80], 'name': ['p', 'q', 'r', 's']}

df\_X = pd.DataFrame(data\_X)

df\_Y = pd.DataFrame(data\_Y)

# Perform inner join

result = pd.merge(df\_X, df\_Y, on='Roll No')

# Print the result

print(result)

**Output:**

**Roll No name\_x name\_y**

**0 10 a p**

**1 20 b q**

**8.** import matplotlib.pyplot as plt

# Data for line graph

data\_line = {'X': [1, 2, 3], 'Y': [10, 15, 12]}

# Data for bar chart

data\_bar = {'Categories': ['A', 'B', 'C'], 'Values': [25, 30, 20]}

# Create a figure with two subplots

fig, axs = plt.subplots(nrows=2, ncols=1, figsize=(8, 6))

# Plot the line graph

axs[0].plot(data\_line['X'], data\_line['Y'], marker='o')

axs[0].set\_xlabel('X')

axs[0].set\_ylabel('Y')

axs[0].set\_title('Line Graph')

# Plot the bar chart

axs[1].bar(data\_bar['Categories'], data\_bar['Values'])

axs[1].set\_xlabel('Categories')

axs[1].set\_ylabel('Values')

axs[1].set\_title('Bar Chart')

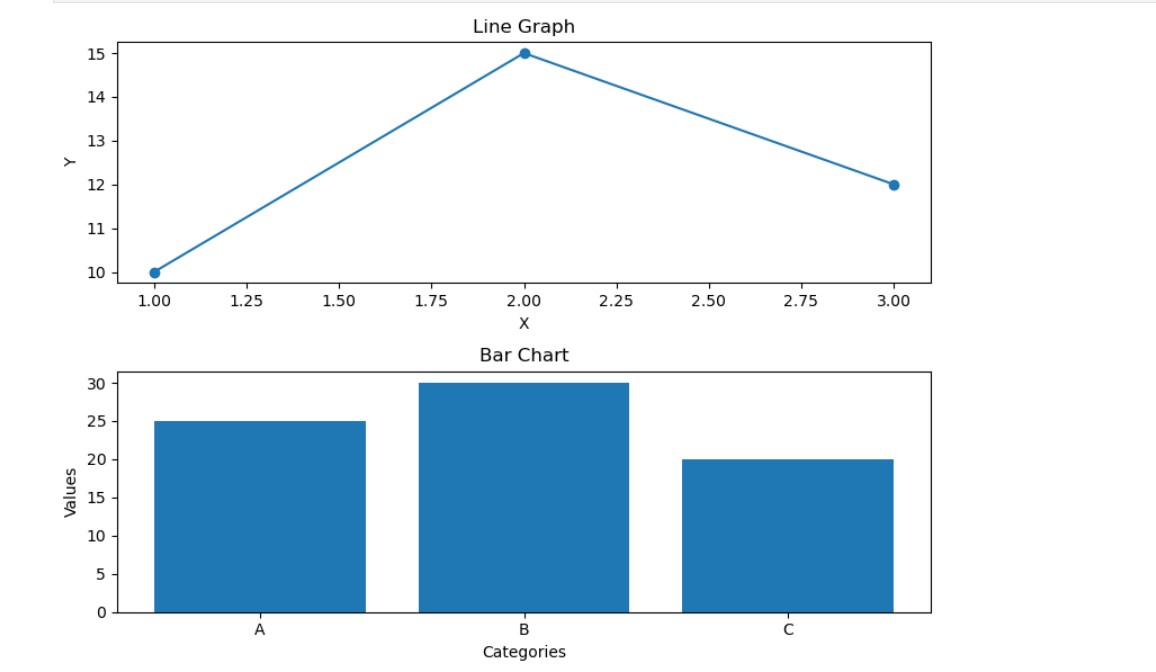
# Adjust spacing between subplots

plt.tight\_layout()

# Show the plot

plt.show()

**Output:**

****

**Section 3:**

2.Kurtosis: A Measure of Tail Distribution:

Kurtosis is a statistical measure that describes the shape of a probability distribution. It quantifies how "heavy-tailed" or "light-tailed" a distribution is compared to a normal distribution.

Types of Kurtosis

Mesokurtic:

A distribution with a kurtosis value similar to that of a normal distribution.

The tails of the distribution are neither too heavy nor too light.

Leptokurtic:

A distribution with a kurtosis value greater than that of a normal distribution.

The tails of the distribution are heavier than those of a normal distribution, meaning there's a higher probability of extreme values.

Examples: Student's t-distribution, Cauchy distribution

Platykurtic:

A distribution with a kurtosis value less than that of a normal distribution.

The tails of the distribution are lighter than those of a normal distribution, meaning there's a lower probability of extreme values.

3. Matplotlib offers a wide range of options for customizing the appearance of plots. Here are five

a)Set the title:

import matplotlib.pyplot as plt

plt.plot([1, 2, 3], [4, 5, 6])

plt.title("My Plot")

plt.show()

b) #axislabels

plt.xlabel("X-axis")

plt.ylabel("Y-axis")

c) #color:

plt.plot([1, 2, 3], [4, 5, 6], color='red')

d) #linestyle:

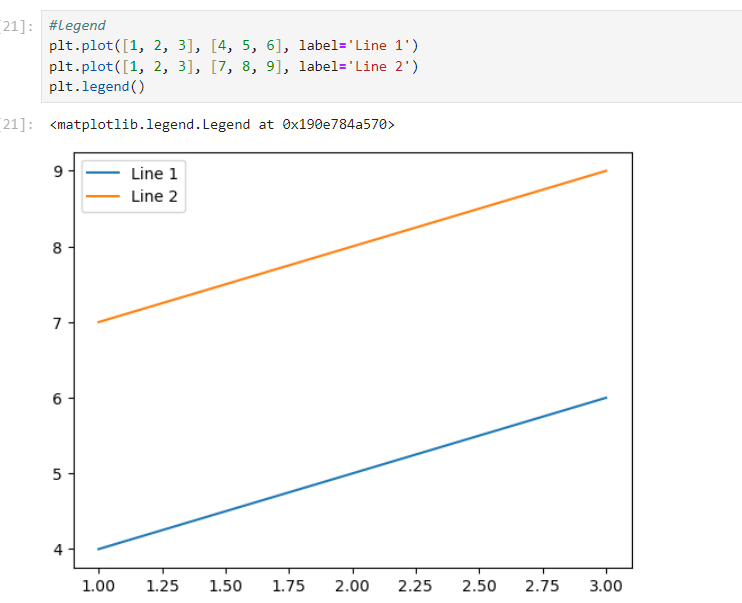
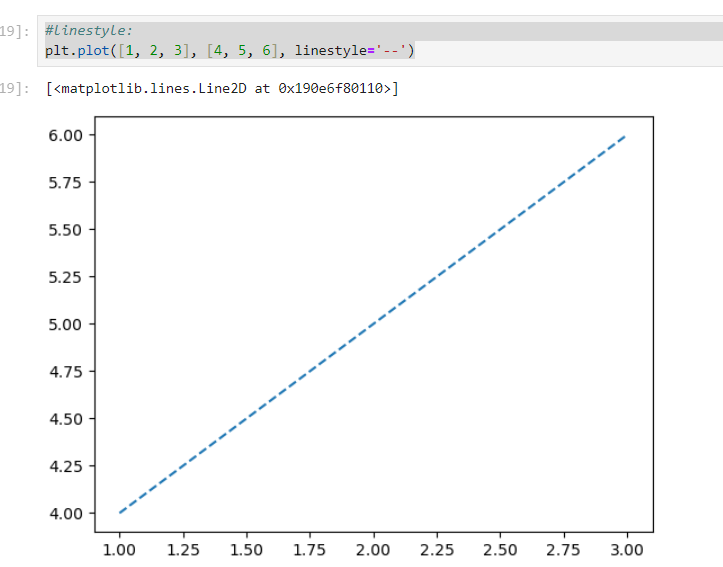
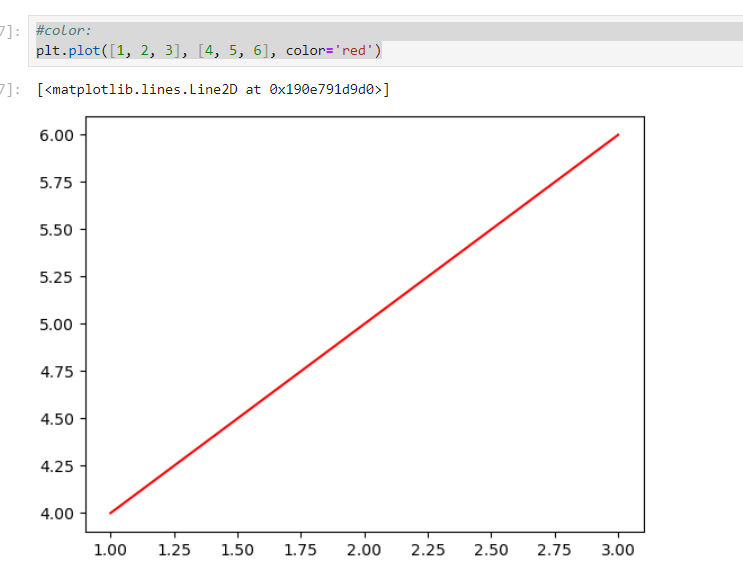
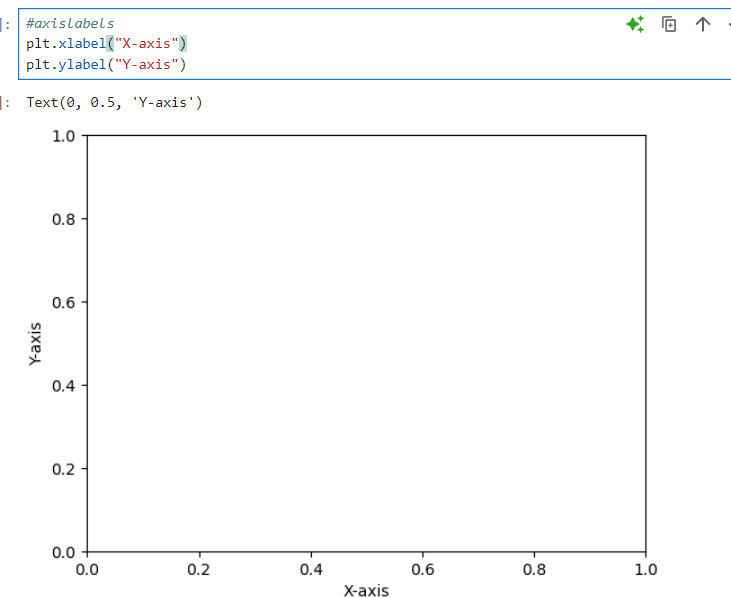
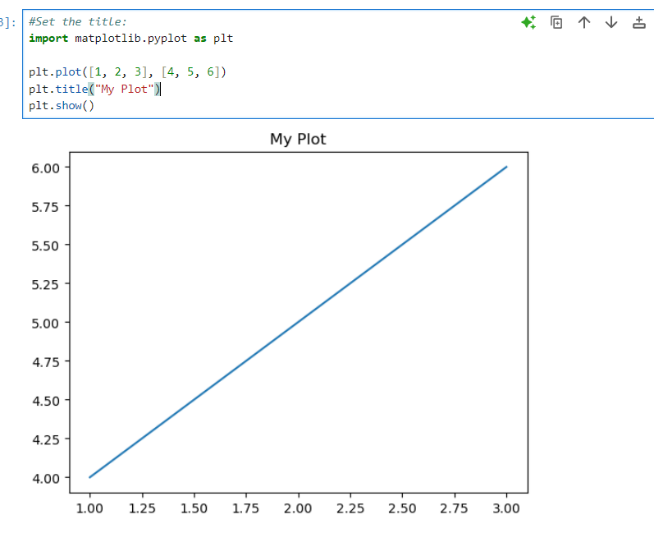
plt.plot([1, 2, 3], [4, 5, 6], linestyle='--')

e) #legend

plt.plot([1, 2, 3], [4, 5, 6], label='Line 1')

plt.plot([1, 2, 3], [7, 8, 9], label='Line 2')

plt.legend()



4. Missing values are a common occurrence in data analysis. It's essential to identify and handle them appropriately to avoid errors in your analysis. Here are three common methods

1. Using the isnull() method:

This method creates a boolean mask where True indicates missing values.

You can use it to count missing values, find their locations, or filter the DataFrame based on missingness.

2. Using the isna() method:

This method is an alias for isnull() and provides the same functionality.

3. Using the dropna() method:

This method removes rows or columns containing missing values.

You can use it to clean your data by removing rows with missing values.

Ex:

# Drop rows with missing values

1.df\_cleaned = df.dropna()

print(df\_cleaned)

2.import pandas as pd

df = pd.DataFrame({'A': [1, 2, None], 'B': ['a', None, 'c']})

missing\_values = df.isnull()

print(missing\_values)

3.import pandas as pd

data = {'A': [1, 2, None, 4],

'B': [5, None, 7, 8],

'C': ['a', 'b', None, 'd']}

df = pd.DataFrame(data)

# Check for missing values

missing\_values = df.isna()

print(missing\_values)

5.

**DataFrame.iloc:**

Integer-based selection: iloc uses integer positions to select data.

Zero-based indexing: The first row and column have index 0.

Slicing: Supports slicing using integer indices.

#Ex:

import pandas as pd

data = {'A': [1, 2, 3], 'B': [4, 5, 6]}

df = pd.DataFrame(data)

# Select the first row

first\_row = df.iloc[0]

print(first\_row)

# Select the second column

second\_column = df.iloc[:, 1]

print(second\_column)

# Select a specific cell

cell\_value = df.iloc[1, 0]

print(cell\_value)

**DataFrame.loc:**

Label-based selection: loc uses labels (row and column names) to select data.

Supports boolean indexing: Can be used with boolean masks to select rows or columns based on conditions.

#ex:

import pandas as pd

data = {'A': [1, 2, 3], 'B': [4, 5, 6]}

df = pd.DataFrame(data, index=['row1', 'row2', 'row3'])

# Select a row by label

row\_by\_label = df.loc['row2']

print(row\_by\_label)

# Select columns by label

columns\_by\_label = df.loc[:, ['A']]

print(columns\_by\_label)

# Select rows based on a condition

filtered\_df = df.loc[df['A'] > 2]

print(filtered\_df)

Major differences:

1.Indexing method: iloc uses integer indices, while loc uses labels.

2.Label-based selection: loc supports label-based selection and boolean indexing, while iloc is strictly integer-based.

3.Slicing behavior: Both can be used for slicing, but iloc uses integer indices, while loc can use labels or boolean masks.

6.

#To convert a Pandas Series to a DataFrame and access specific data elements, follow these steps:

import pandas as pd

series = pd.Series([10, 20, 30, 40], index=['A', 'B', 'C', 'D'])

#Convert the Series to a DataFrame:

df = series.to\_frame()

#This will create a DataFrame with a single column, where the Series index becomes the column name and the Series values become the column data.

#Access specific data elements:

#By row and column index:

value = df.iloc[1, 0] # Access the element at the second row and first column

print(value)

#By row and column label:

value = df.loc['B', 0] # Access the element at the row labeled 'B' and the first column

print(value)

#By column name:

column\_data = df[0] # Access the data from the first column as a Series

print(column\_data)

#By row label:

row\_data = df.loc['C'] # Access the data from the row labeled 'C' as a Series

print(row\_data)

#example:

import pandas as pd

series = pd.Series([10, 20, 30, 40], index=['A', 'B', 'C', 'D'])

df = series.to\_frame()

# Accessing specific data elements

value1 = df.iloc[2, 0]

value2 = df.loc['C', 0]

column\_data = df[0]

row\_data = df.loc['B']

print(value1)

print(value2)

print(column\_data)

print(row\_data)

**Output:**

20

20

A 10

B 20

C 30

D 40

Name: 0, dtype: int64

0 30

Name: C, dtype: int64

30

30

A 10

B 20

C 30

D 40

Name: 0, dtype: int64

0 20

Name: B, dtype: int64

**7.** #To create a Pandas Series from a Python list with custom index values, you can use the following steps:

#Import the Pandas library:

import pandas as pd

#Create a Python list:

my\_list = [10, 20, 30, 40]

#Create a custom index:

my\_index = ['A', 'B', 'C', 'D']

#Create a Pandas Series:

my\_series = pd.Series(my\_list, index=my\_index)

#This will create a Pandas Series with the elements from the my\_list list and the corresponding custom index values from the my\_index list.

print(my\_series)

Output:

A 10

B 20

C 30

D 40

dtype: int64

Section:2

1. Python lists vs. NumPy arrays:

Python lists: General-purpose data structures for storing arbitrary objects. Slower for numerical operations due to dynamic typing.

NumPy arrays: Optimized for numerical operations on homogeneous data. Faster and more efficient for scientific computing.

2. # Indexing and slicing in NumPy arrays:

#Indexing: Accessing individual elements using integer positions (0-based).

#Slicing: Extracting a subset of elements using a range of indices.

import numpy as np

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

# Indexing

first\_element = arr[0] # Access the first element

last\_element = arr[-1] # Access the last element

# Slicing

subarray = arr[1:4] # Extract elements from index 1 to 3 (exclusive)

3. #Pandas Series vs. NumPy array:

#Pandas Series: One-dimensional labeled array with potential for missing values and custom indexing.

#NumPy array: Fixed-size, homogeneous array with no built-in labels.

import numpy as np

# Create a NumPy array

num\_array = np.array([10, 20, 30])

# Access elements by integer index

first\_element = num\_array[0] # Output: 10

import pandas as pd

# Create a Pandas Series with custom index

series = pd.Series([10, 20, 30], index=['A', 'B', 'C'])

# Access elements by index label

value\_by\_label = series['B'] # Output: 20

# Access elements by integer index

value\_by\_position = series[1] # Output: 20

4. Accessing elements in a Pandas Series:

#By label: Using the index label.

#By position: Using integer-based indexing.

import pandas as pd

series = pd.Series([10, 20, 30], index=['A', 'B', 'C'])

# By label

value\_by\_label = series['B']

# By position

value\_by\_position = series[1]

5:The read\_csv() function in pandas is used to read data from a CSV (Comma-Separated Values) file and create a pandas DataFrame object. This DataFrame represents the data in a tabular format, making it easy to manipulate and analyze.

#Key parameters of the read\_csv() function:

#header: The row number to use as the column names (default: 0).

#names: A list of column names to use instead of the column names in the CSV file.

#dtype: A dictionary specifying the data type for each column.

#index\_col: The column to use as the index of the DataFrame.

#Ex:

import pandas as pd

# Read a CSV file into a DataFrame

df = pd.read\_csv('data.csv')

# Print the DataFrame

print(df)

8:Matplotlib is a powerful library in Python used for creating static, animated, and interactive visualizations. It provides a flexible and customizable interface for creating various types of plots, including line plots, bar charts, histograms, scatter plots, and more. Matplotlib is essential for data scientists to visually explore and understand data trends, patterns, and relationships.

The pyplot module within Matplotlib provides a high-level interface that mimics MATLAB's plotting functions. This makes it easy for users familiar with MATLAB to transition to Python for data visualization. The pyplot module simplifies the process of creating plots by providing functions for setting plot parameters, adding labels, titles, and legends, and customizing the appearance of plots.

9:

Line graph: Best suited for visualizing trends over time or relationships between two variables.

Pie chart: Effective for representing categorical data as proportions of a whole.

Vertical bar chart: Ideal for comparing values across different categories.

Horizontal bar chart: Similar to vertical bar charts, but can be useful when categories have long names or when you want to emphasize the values.

10:

Difference between fetchone() and fetchall()

fetchone(): Fetches the next row from a database cursor. Returns a single row as a tuple.

fetchall(): Fetches all remaining rows from a database cursor. Returns a list of tuples, where each tuple represents a row.

Section1:

1.c. Numpy

2.d. len()

3.c. Matplotlib

4.d. sns.catplot()

5.a. plt.scatter()

6.a. mysql-connector

7.c. Arrays have a fixed size and cannot be resized after creation, while lists are dynamic and can change in size.

8.c. df.drop("Age", axis=1)