**✅ Question 67:**

**Write a Python program to create a text file, insert n lines of text and display its contents in a neat format. Also, count the number of lines, words, and characters in the text file.**  
**Explain in detail with an example, line by line.**

**✅ Answer:**

**💻 Python Program:**

# Step 1: Create a text file and write n lines to it

filename = "sample\_text.txt"

n = int(input("Enter the number of lines to write in the file: "))

# Opening the file in write mode

with open(filename, "w") as file:

print(f"\nEnter {n} lines of text:")

for i in range(n):

line = input(f"Line {i+1}: ")

file.write(line + "\n") # Add newline after each line

# Step 2: Read the file and display its contents

print("\n--- File Contents ---")

with open(filename, "r") as file:

contents = file.read()

print(contents)

# Step 3: Count number of lines, words and characters

with open(filename, "r") as file:

lines = file.readlines()

line\_count = len(lines)

word\_count = sum(len(line.split()) for line in lines)

char\_count = sum(len(line) for line in lines)

# Display counts

print("Statistics:")

print(f"Number of lines : {line\_count}")

print(f"Number of words : {word\_count}")

print(f"Number of characters: {char\_count}")

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**✅ Question 68:**

**Provide examples demonstrating how to open, read, and write to text files in Python using built-in functions and methods. Discuss common file modes and their implications in file handling operations.**   
**(Explain in detail with example line by line)**

**✅ Answer:**

Python provides powerful built-in functions to handle text files. The most commonly used function is:

open(filename, mode)

Where:

* filename is the name of the file
* mode defines how the file should be opened (read, write, append, etc.)

**🔁 Common File Modes in Python**

| **Mode** | **Description** |
| --- | --- |
| 'r' | Read mode (default). File must exist. |
| 'w' | Write mode. Creates a new file or overwrites existing one. |
| 'a' | Append mode. Adds to end of file, without deleting existing content. |
| 'r+' | Read and write mode. File must exist. |
| 'w+' | Write and read. Overwrites file or creates new one. |
| 'a+' | Append and read. Creates file if it doesn't exist. |
| 'b' | Binary mode (used with other modes, like rb, wb) |

**✅ Examples with Line-by-Line Explanation**

**✍️ 1. Writing to a File ('w' mode)**

# Open file in write mode

with open("example.txt", "w") as file:

file.write("Hello, Python!\n")

file.write("This is a file-handling example.")

**Explanation:**

* open("example.txt", "w"): Opens the file in write mode. Creates file if not existing.
* with statement automatically closes the file.
* file.write(...): Writes string data to the file.
* \n ensures the next line is on a new line.

**📖 2. Reading from a File ('r' mode)**

# Open file in read mode

with open("example.txt", "r") as file:

content = file.read()

print("File content:\n", content)

**Explanation:**

* open("example.txt", "r"): Opens the file in read mode.
* file.read(): Reads the entire content as a string.
* print(...): Displays content on console.

**📄 3. Appending to a File ('a' mode)**

# Open file in append mode

with open("example.txt", "a") as file:

file.write("\nLet's add more text.")

**Explanation:**

* open("example.txt", "a"): Opens file in append mode.
* Appends text at the end of the file without deleting previous content.

**📋 4. Reading Line by Line**

# Open file and read line by line

with open("example.txt", "r") as file:

print("Reading line by line:")

for line in file:

print(line.strip())

**Explanation:**

* for line in file: Iterates over each line in the file.
* strip(): Removes newline characters and extra spaces from ends.

**🔁 5. Using readline() and readlines()**

with open("example.txt", "r") as file:

first\_line = file.readline()

all\_lines = file.readlines()

print("First line:", first\_line.strip())

print("Remaining lines:", all\_lines)

**Explanation:**

* readline(): Reads one line at a time.
* readlines(): Returns all lines as a list of strings.

**🧠 Why Use with open(...) Instead of open() Alone?**

with open("file.txt", "r") as f:

...

* It automatically **closes the file** when the block is done.
* Safer and cleaner way to handle files.
* Prevents memory leaks and file corruption.

**📌 Real-World Use Case Example**

# Create a log file that stores user activity

with open("user\_log.txt", "a") as log:

log.write("User logged in at 10:00 AM\n")

This example shows how logs are maintained using append mode.

**✅ Conclusion**

* Python file handling is efficient using built-in methods like open(), read(), write(), and close().
* Using appropriate file modes (r, w, a, etc.) is crucial for correct operation.
* The with statement is the recommended approach to ensure files are properly managed.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Here’s a **detailed 10-mark answer** for **Question 69**, complete with explanation, features, and example of **PyTables**, line-by-line.

**✅ Question 69:**

**What is PyTables? Explain its purpose and key features (explain in detail with example line by line).**

**✅ Answer:**

**📘 What is PyTables?**

**PyTables** is a **Python library** used to **manage large amounts of structured data** in a hierarchical way using the **HDF5 (Hierarchical Data Format version 5)** file format.

It is designed for **efficiency** (fast read/write), **scalability**, and **complex data querying** — especially suitable for large datasets used in data science, machine learning, or scientific computing.

**🎯 Purpose of PyTables**

* **Efficient storage** of large datasets (millions of records).
* **Fast querying** and retrieval of complex or hierarchical data.
* Acts as a **bridge between NumPy arrays** and **disk-based storage**.
* Used in **research**, **finance**, **big data**, and **scientific applications**.

**✨ Key Features of PyTables**

| **Feature** | **Description** |
| --- | --- |
| **HDF5-based** | Stores data in highly structured, binary format with compression. |
| **Hierarchical** | Supports tree-like structure with groups and datasets. |
| **Compression** | Reduces file size with zlib, bzip2, etc. |
| **Querying** | Enables fast, SQL-like condition queries on datasets. |
| **Supports NumPy** | Seamlessly works with NumPy arrays. |
| **Portable** | HDF5 files can be shared across platforms and languages (e.g., MATLAB, R). |

**✅ Line-by-Line Example: Using PyTables**

**▶️ Install PyTables First (if not already installed):**

pip install tables

**💻 Python Program Using PyTables**

import tables

import numpy as np

# Step 1: Define a data structure using tables.IsDescription

class Student(tables.IsDescription):

name = tables.StringCol(16) # 16-character string

age = tables.Int32Col() # 32-bit integer

marks = tables.Float64Col() # 64-bit float

**Explanation:**

* tables.IsDescription: Used to define a structured table schema.
* StringCol, Int32Col, etc.: Define column types (like database fields).

# Step 2: Create an HDF5 file and a table inside it

file = tables.open\_file("students.h5", mode="w", title="Student Data")

# Create a group and then a table inside that group

group = file.create\_group("/", 'school', 'School Group')

table = file.create\_table(group, 'students', Student, 'Student Info Table')

student = table.row

**Explanation:**

* open\_file: Creates a new HDF5 file.
* create\_group: Makes a folder-like group called /school.
* create\_table: Creates a table structure using the Student schema.

# Step 3: Insert data into the table

student['name'] = 'Alice'

student['age'] = 20

student['marks'] = 85.5

student.append()

student['name'] = 'Bob'

student['age'] = 22

student['marks'] = 91.2

student.append()

# Save the data

table.flush()

**Explanation:**

* Each student['column'] = value assigns data to a row.
* append() adds it to the table.
* flush() writes all pending changes to disk.

# Step 4: Read and display records

print("All Student Records:")

for row in table:

print(f"{row['name'].decode()} | Age: {row['age']} | Marks: {row['marks']}")

**Explanation:**

* Loops through each row and prints.
* decode() is used to convert bytes to string.

# Step 5: Query the table for students with marks > 90

print("\nStudents with Marks > 90:")

for row in table.where('marks > 90'):

print(f"{row['name'].decode()} | Marks: {row['marks']}")

**Explanation:**

* .where('marks > 90') is similar to SQL querying.
* Only displays students who scored more than 90.

file.close()

* **Closes the file** to save changes and free memory.

**📄 Sample Output:**

All Student Records:

Alice | Age: 20 | Marks: 85.5

Bob | Age: 22 | Marks: 91.2

Students with Marks > 90:

Bob | Marks: 91.2

**✅ Conclusion**

* **PyTables** is a powerful tool for **structured storage** and **fast access** to large-scale data.
* It combines the **speed of C**, the **structure of SQL**, and the **flexibility of Python**.
* Ideal for use cases like **data warehousing**, **sensor data logging**, **machine learning preprocessing**, and more.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Here is a **simple and detailed 10-mark answer** for **Question 70**, including the **use case of PyTables** and a **basic example** showing how to create a PyTable, store data, and read data — step-by-step, suitable for exams.

**✅ Question 70:**

**Explain the use case of PyTables and also show how to create PyTable, store data and read data.**  
*(With simple example and explanation)*

**✅ Answer:**

**📘 What is PyTables?**

**PyTables** is a Python library used to **store and manage large datasets efficiently** using the **HDF5 file format**. It is ideal for applications that require:

* Fast access to large structured datasets
* Storing tabular or hierarchical data
* Searching and filtering data efficiently

**🎯 Use Cases of PyTables**

1. **Scientific Data Storage:**  
   Researchers use it to store data from experiments or simulations.
2. **Sensor and IoT Data Logging:**  
   PyTables can handle time-series data collected from sensors.
3. **Machine Learning Datasets:**  
   Large training datasets can be stored and retrieved efficiently.
4. **Big Data Applications:**  
   PyTables helps manage millions of records with fast query performance.

**💻 Simple Example: Create, Store, and Read PyTables**

**✅ Step 1: Install PyTables (if not already installed)**

pip install tables

**✅ Step 2: Python Code (With Line-by-Line Explanation)**

import tables

import numpy as np

# Step 1: Define a table structure (schema)

class Person(tables.IsDescription):

name = tables.StringCol(16) # String column (max 16 characters)

age = tables.Int32Col() # Integer column

score = tables.Float32Col() # Float column

**Explanation:**

* A class Person defines the structure of our table.
* It has three fields: name, age, and score.

# Step 2: Create an HDF5 file and a table

file = tables.open\_file("people.h5", mode="w", title="People Info")

table = file.create\_table("/", "person\_table", Person, "Person Data")

person = table.row # Reference to add new rows

**Explanation:**

* open\_file: Creates a new file named people.h5.
* create\_table: Adds a table named person\_table based on the Person schema.

# Step 3: Add simple data

person['name'] = 'Alice'

person['age'] = 25

person['score'] = 88.5

person.append()

person['name'] = 'Bob'

person['age'] = 30

person['score'] = 92.0

person.append()

table.flush() # Save changes

**Explanation:**

* Each person['field'] = value assigns data.
* append() adds the row to the table.
* flush() writes everything to disk.

# Step 4: Read and display data

print("Reading data from PyTable:")

for row in table:

print(f"{row['name'].decode()} | Age: {row['age']} | Score: {row['score']}")

**Explanation:**

* Loops through all rows in the table.
* decode() converts stored byte strings back to normal strings.

file.close() # Close the file when done

**📄 Output:**

Reading data from PyTable:

Alice | Age: 25 | Score: 88.5

Bob | Age: 30 | Score: 92.0

**📌 Summary**

| **Step** | **Action** |
| --- | --- |
| 1 | Define a table structure using a class |
| 2 | Create an HDF5 file and a table |
| 3 | Store data (rows) into the table |
| 4 | Read and display the stored data |

**✅ Conclusion**

* PyTables is **easy to use**, even for beginners.
* It's ideal for working with **structured data** that grows over time.
* The example above shows how to **create**, **store**, and **read** data with minimal code.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Here’s a **detailed and exam-ready 10-mark answer** for **Question 71**, including clear explanation and **simple examples** for creating **Pandas Series** and **DataFrames**, suitable for beginners.

**✅ Question 71:**

**How do you create Pandas Series and DataFrame? Explain with the help of examples.**

**✅ Answer:**

**📚 What is Pandas?**

**Pandas** is a powerful Python library used for **data manipulation and analysis**. It provides two primary data structures:

1. **Series** – One-dimensional labeled array.
2. **DataFrame** – Two-dimensional labeled table (like a spreadsheet or SQL table).

**✅ 1. Creating a Pandas Series**

A **Series** is like a one-column list with labels (called index).

**📌 Example 1: Create Series from a List**

import pandas as pd

# Create a Series from a list

data = [10, 20, 30, 40]

s = pd.Series(data)

print("Pandas Series:")

print(s)

**🧾 Output:**

0 10

1 20

2 30

3 40

dtype: int64

**✅ Explanation:**

* pd.Series(data) creates a Series.
* Numbers on the left (0, 1, 2...) are **indexes**.
* You can **customize the index**:

s = pd.Series([10, 20, 30], index=['a', 'b', 'c'])

print(s)

**Output:**

a 10

b 20

c 30

dtype: int64

**✅ 2. Creating a Pandas DataFrame**

A **DataFrame** is like a table with **rows and columns**.

**📌 Example 2: Create DataFrame from Dictionary**

# Create DataFrame from dictionary

data = {

'Name': ['Alice', 'Bob', 'Charlie'],

'Age': [25, 30, 35],

'Score': [85.5, 90.0, 92.5]

}

df = pd.DataFrame(data)

print("Pandas DataFrame:")

print(df)

**🧾 Output:**

Name Age Score

0 Alice 25 85.5

1 Bob 30 90.0

2 Charlie 35 92.5

**📌 Example 3: Create DataFrame from List of Lists**

data = [

['Alice', 25],

['Bob', 30],

['Charlie', 35]

]

df = pd.DataFrame(data, columns=['Name', 'Age'])

print(df)

**Output:**

Name Age

0 Alice 25

1 Bob 30

2 Charlie 35

**📌 Example 4: Add Index Manually**

df = pd.DataFrame(data, columns=['Name', 'Age'], index=['a', 'b', 'c'])

print(df)

**Output:**

Name Age

a Alice 25

b Bob 30

c Charlie 35

**✅ Summary of Functions**

| **Function** | **Purpose** |
| --- | --- |
| pd.Series(data) | Creates a Series |
| pd.DataFrame(data) | Creates a DataFrame |
| columns=[] | Specifies column names |
| index=[] | Specifies custom row labels |

**✅ Conclusion**

* A **Series** is a simple one-dimensional array with labels.
* A **DataFrame** is a table with rows and columns, similar to Excel.
* Both structures are very useful for **data analysis** and **processing** in Python.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**✅ Question 72:**

**With reference to Pandas explain:**

**i) info()  
ii) describe()  
iii) head()  
iv) columns**

**✅ Answer:**

**📚 Introduction to Pandas DataFrame Methods**

Pandas provides several **built-in methods** to help you **understand and explore your data**. Below are four commonly used methods:

**✅ i) info()**

**📌 Purpose:**

Gives a **summary** of the DataFrame — including **column names**, **data types**, **non-null values**, and **memory usage**.

**📌 Example:**

import pandas as pd

data = {

'Name': ['Alice', 'Bob', 'Charlie'],

'Age': [25, 30, 35],

'Score': [85.0, 90.5, 92.3]

}

df = pd.DataFrame(data)

df.info()

**🧾 Output:**

<class 'pandas.core.frame.DataFrame'>

RangeIndex: 3 entries, 0 to 2

Data columns (total 3 columns):

# Column Non-Null Count Dtype

--- ------ -------------- -----

0 Name 3 non-null object

1 Age 3 non-null int64

2 Score 3 non-null float64

dtypes: float64(1), int64(1), object(1)

memory usage: 200.0 bytes

**✅ Explanation:**

* Shows the number of **rows and columns**
* Displays **data types** of each column
* Shows how many values are **non-null**

**✅ ii) describe()**

**📌 Purpose:**

Gives **statistical summary** of numerical columns — like **count, mean, std, min, max, quartiles**.

**📌 Example:**

df.describe()

**🧾 Output:**

Age Score

count 3.000000 3.00000

mean 30.000000 89.26667

std 5.000000 3.70106

min 25.000000 85.00000

25% 27.500000 87.75000

50% 30.000000 90.50000

75% 32.500000 91.40000

max 35.000000 92.30000

**✅ Explanation:**

* Useful to understand **distribution of numerical data**
* Quickly shows **mean**, **median**, **standard deviation**, etc.

**✅ iii) head()**

**📌 Purpose:**

Displays the **first 5 rows** of the DataFrame by default.

**📌 Example:**

df.head()

**🧾 Output:**

Name Age Score

0 Alice 25 85.0

1 Bob 30 90.5

2 Charlie 35 92.3

**✅ Explanation:**

* Used for **quick preview** of data.
* You can pass a number to head(n) to show the first n rows.

df.head(2) # Shows first 2 rows

**✅ iv) columns**

**📌 Purpose:**

Returns the list of **column names** in the DataFrame.

**📌 Example:**

print(df.columns)

**🧾 Output:**

Index(['Name', 'Age', 'Score'], dtype='object')

**✅ Explanation:**

* Helps in checking or renaming column names.
* You can convert it to a list:

print(list(df.columns))

# Output: ['Name', 'Age', 'Score']

**🧠 Summary Table**

| **Method** | **Purpose** | **Useful For** |
| --- | --- | --- |
| info() | Summary of structure and data types | Quick overview of dataset |
| describe() | Statistics for numeric columns | Data analysis and summary stats |
| head() | First few rows of data | Previewing large datasets |
| columns | Shows column names | Accessing or modifying columns |

**✅ Conclusion**

These Pandas methods are essential for:

* Exploring and understanding data
* Preparing for cleaning or analysis
* Getting quick insights without manual inspection

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**✅ Question 73:**

**What are missing values in a dataset? How can you identify and fill missing values using Pandas in Python? Explain any four methods to handle missing values.**

**✅ Answer:**

**📘 What are Missing Values?**

**Missing values** in a dataset are the **empty or null entries** where data is not available. They can be caused by:

* Human error during data entry
* Incomplete data collection
* Corrupted files
* Data not applicable in some cases

In **Pandas**, missing values are usually represented as:

NaN (Not a Number)

None

**🔍 Identifying Missing Values in Pandas**

To find missing values in a DataFrame:

import pandas as pd

import numpy as np

# Example DataFrame

data = {

'Name': ['Alice', 'Bob', 'Charlie', 'David'],

'Age': [25, np.nan, 30, None],

'Score': [85, 90, np.nan, 88]

}

df = pd.DataFrame(data)

# Identify missing values

print(df.isnull())

**🧾 Output:**

Name Age Score

0 False False False

1 False True False

2 False False True

3 False True False

* True means **missing value**
* False means **data is present**

**✅ Four Methods to Handle Missing Values**

**✅ 1. Remove Rows with Missing Data**

df\_cleaned = df.dropna()

print(df\_cleaned)

**Explanation:**

* dropna() removes **rows** that contain **any missing values**
* Use when **missing data is small** and **not critical**

**✅ 2. Fill Missing Values with a Constant**

df\_filled = df.fillna(0)

print(df\_filled)

**Explanation:**

* fillna(0) replaces all NaN with 0
* Useful for numeric columns like scores or counts

**✅ 3. Fill with Mean or Median**

mean\_age = df['Age'].mean()

df['Age'] = df['Age'].fillna(mean\_age)

**Explanation:**

* Replaces missing values with the **average** of the column
* Good for **numerical data** where mean makes sense

**✅ 4. Forward Fill (Propagate previous value)**

df\_ffill = df.fillna(method='ffill')

print(df\_ffill)

**Explanation:**

* ffill fills missing value with **previous row’s value**
* Useful in **time series** or **sequential data**

**✅ Summary Table**

| **Method** | **Function** | **Description** |
| --- | --- | --- |
| Drop rows | dropna() | Removes rows with missing values |
| Fill with constant | fillna(0) | Replaces NaN with a fixed value |
| Fill with mean | fillna(mean) | Replaces NaN with column average |
| Forward fill | fillna(method='ffill') | Fills using previous value |

**✅ Conclusion**

* **Missing values** are common in real-world data.
* Pandas provides **easy methods** to detect and fix them.
* Handling missing data is important for **accurate analysis** and **machine learning**.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Here’s a **clear, detailed 10-mark answer** for **Question 74**, with simple explanation and examples using Python and Pandas to demonstrate **aggregate functions with GROUP BY** on sales data grouped by salesman.

**✅ Question 74:**

**Explain any four aggregate functions commonly used with the GROUP BY clause. Illustrate each with an example using a Python dictionary of sales data, where you group the data by salesman.**

**✅ Answer:**

**📘 What is GROUP BY?**

* The **GROUP BY** clause is used to **group rows** in a dataset based on a column (here, salesman), so that aggregate functions can be applied on each group.
* Common in SQL and Pandas for summarizing data.

**✅ Four Common Aggregate Functions**

1. **SUM()** – Adds up all values in a group.
2. **COUNT()** – Counts the number of entries in a group.
3. **MAX()** – Finds the maximum value in a group.
4. **AVG() (mean)** – Calculates the average value in a group.

**📊 Example: Sales Data**

import pandas as pd

# Sample sales data in a dictionary

sales\_data = {

'salesman': ['Alice', 'Bob', 'Alice', 'Bob', 'Charlie', 'Alice'],

'sales': [250, 150, 300, 200, 400, 100]

}

# Create DataFrame

df = pd.DataFrame(sales\_data)

print("Original Data:")

print(df)

**Output:**

salesman sales

0 Alice 250

1 Bob 150

2 Alice 300

3 Bob 200

4 Charlie 400

5 Alice 100

**✅ 1. SUM() – Total Sales by Salesman**

sum\_sales = df.groupby('salesman')['sales'].sum()

print("\nTotal Sales by Salesman:")

print(sum\_sales)

**Output:**

salesman

Alice 650

Bob 350

Charlie 400

Name: sales, dtype: int64

**Explanation:** Adds all sales for each salesman.

**✅ 2. COUNT() – Number of Sales Entries per Salesman**

count\_sales = df.groupby('salesman')['sales'].count()

print("\nCount of Sales per Salesman:")

print(count\_sales)

**Output:**

salesman

Alice 3

Bob 2

Charlie 1

Name: sales, dtype: int64

**Explanation:** Counts how many sales records each salesman has.

**✅ 3. MAX() – Highest Sale Made by Each Salesman**

max\_sale = df.groupby('salesman')['sales'].max()

print("\nMaximum Sale by Salesman:")

print(max\_sale)

**Output:**

salesman

Alice 300

Bob 200

Charlie 400

Name: sales, dtype: int64

**Explanation:** Finds the biggest single sale for each salesman.

**✅ 4. AVG() (Mean) – Average Sale by Salesman**

avg\_sale = df.groupby('salesman')['sales'].mean()

print("\nAverage Sale by Salesman:")

print(avg\_sale)

**Output:**

salesman

Alice 216.666667

Bob 175.000000

Charlie 400.000000

Name: sales, dtype: float64

**Explanation:** Calculates the average sales amount for each salesman.

**✅ Summary Table**

|  |  |  |
| --- | --- | --- |
| Aggregate Function | Function Used in Pandas | Purpose |
| SUM | .sum() | Total of values in each group |
| COUNT | .count() | Number of entries in each group |
| MAX | .max() | Maximum value in each group |
| AVG (mean) | .mean() | Average value in each group |

**✅ Conclusion**

* Using **GROUP BY** with aggregate functions helps **summarize data easily**.
* Pandas makes it simple to perform these operations on grouped data.
* These functions are widely used in sales, finance, and many other data analysis tasks.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*xx\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*