🐼 The Ultimate Pandas Guide: From Zero to Data Wizard

with Real-World Examples & Hands-on Practice

*Created by **Goutam Kuiri** | 🔗 LinkedIn*

💌 Want the full Jupyter Notebook? DM me — I'll be happy to share it with you! 🚀

1. 🚀 Pandas Basics – Build a Strong Foundation

🐼 What is Pandas?

📖 English:

Pandas is an **open-source Python library** mainly used for **data manipulation and analysis**. It provides two key data structures: Series (1-D) and DataFrame (2-D), which make it easy to handle structured data like tables. It is widely used for cleaning, analyzing, and visualizing datasets.

🗣️ Hinglish:

Pandas ek **Python library** hai jo **data ko handle aur analyze** karne ke liye use hoti hai. Isme do main structures hote hain: Series (1-D, ek column jaisa) aur DataFrame (2-D, ek table jaisa). Iske through hum easily data clean, transform aur visualize kar sakte hain.

🔹 Why Use Pandas?

📖 English:

Pandas makes data handling simple and efficient.

• Allows you to clean, transform, merge, filter, and analyze datasets quickly.

• Saves time and effort compared to raw Python code.

• Highly popular in data science, machine learning, and analytics projects.

🗣️ Hinglish:

Pandas data ko handle karna **easy aur fast** banata hai.

- Iske through hum **data clean, transform, merge, filter** aur analyze kar sakte hain.
- Normal Python code ke comparison me **Pandas** kaafi time aur effort bachata hai.
- Ye **data science, machine learning** aur real-world analytics me bohot use hota hai.

🔹 What are Data Structures in Pandas?

📖 English Definition:

In **Pandas**, data structures are the built-in containers that hold and organize data efficiently for analysis. They give shape to the data, making it easy to manipulate, clean, transform, and analyze. Pandas mainly provides two core data structures:

- **Series (1D):** A one-dimensional labeled array, like a single column.
- **DataFrame (2D):** A two-dimensional labeled structure, like a full table with rows & columns.

Basically, these structures are the **backbone of Pandas** — without them, Pandas wouldn't be Pandas.

🗣️ Hinglish Definition:

Pandas me data structures matlab woh containers jo data ko organize aur store karte hain ek proper shape me, taaki analysis easy ho jaye. Ye ek tarah ka **backbone** hai Pandas ka.

- **Series (1D):** Ek single column jaisa hota hai, labeled array.
- **DataFrame (2D):** Ek poori table jaisa hota hai jisme rows aur columns dono hote hain.

Soch lo, **Series = Excel ka single column** aur **DataFrame = poora Excel sheet** 📊

(i) 📌 Pandas Series (1D)

📖 English:

A **Pandas Series** is like a **single column of data**. You can think of it as a **one-dimensional array** that can store **different types of data** (integers, floats, strings, etc.). What makes it powerful is that each value has a **label (index)** attached to it, so you don't just access data by **position** but also by **name**.

👉 *Example:* A column of **student names**, or a column of their **marks**.

🗣️ Hinglish:

Pandas Series ek **single column** jaisa hota hai. Ye basically ek one-dimensional array hai jisme **alag-alag type ka data** store kar sakte ho (integers, floats, strings, etc.). Har value ke saath ek index (label) hota hai, matlab tum data ko sirf **position** se hi nahi, balki **naam** se bhi access kar sakte ho.

👉 *Example:* Ek column me sirf **students ke naam**, ya ek column me sirf unke **marks**.

(ii) 📌 Pandas DataFrame (2D)

📖 English:

A Pandas DataFrame is like a **full table of data**. It is a two-dimensional structure with **rows and columns**, where each column is like a **Series**. DataFrames make it easy to handle structured datasets just like an Excel sheet or SQL table.

👉 *Example:* A table with **student names** in one column and their **marks** in another.

🗣️ Hinglish:

Pandas DataFrame ek **poori table** jaisa hota hai. Ye basically ek two-dimensional structure hai jisme **rows aur columns** hote hain, aur har column ek **Series** ki tarah hota hai. Iske through tum Excel sheet ya SQL table jaisa structured data handle kar sakte ho.

👉 *Example:* Ek table jisme ek column me **students ke naam** aur dusre me **marks**.

(iii) 📌 Series vs DataFrame (Comparison)

📖 English:

Here's a simple comparison between Series and DataFrame to understand them better:

Feature	Series (1D)	DataFrame (2D)
Dimension	1-D	2-D
Structure	Single column (like a vector)	Table with rows & columns

Data Types	Holds one type at a time	Can hold multiple types (int, float, string, etc.)
Example	[85, 90, 78]	{"Name": ["A", "B"], "Marks": [85, 90]}

🗣️ Hinglish:

Chhota sa comparison dekh lo **Series** aur **DataFrame** ke beech:

Feature	**Series (1D)**	**DataFrame (2D)**
Dimension	**1-D**	**2-D**
Structure	Ek single column	Ek poori table (rows + columns)
Data Types	Ek hi type ka data	Multiple types (int, float, string, etc.)
Example	[85, 90, 78]	{"Name": ["A", "B"], "Marks": [85, 90]}

✅ What is a Pandas Series (1D)?

📖 English:

In **Pandas**, a **Series** is a **one-dimensional labeled array** that can hold data of any type (integers, strings, floats, objects, etc.).

- It is like a **column in Excel** or a **1D NumPy array**, but with **labels (index)**.

🗣️ Hinglish:

Pandas me **Series** ek **1D labeled array** hai jo kisi bhi type ka data store kar sakta hai (int, float, string, etc.).

- Ye **Excel ke column** jaisa hota hai, jisme **index (labels)** hote hain.

🔹 Syntax:

```
import pandas as pd
```

```
pd.Series(data=None, index=None, dtype=None, name=None,
copy=False)
```

- **data** → Input data (list, NumPy array, dict, scalar, etc.)
- **index** → Labels for elements (optional, default = 0,1,2,...)
- **dtype** → Data type (optional)
- **name** → Name of the Series (optional)
- **Returns** → Pandas Series object

📌 Real-Life Examples of Pandas Series

🍎 1. Create Series from List

In [4]:

```
import pandas as pd

s = pd.Series([10,20,30,45,60])
print(s)
```

```
0    10
1    20
2    30
3    45
4    60
dtype: int64
```

📝 Hinglish: List ko Series banaya, default index mila

🚗 2. Create Series with Custom Index

In [6]:

```
s = pd.Series([100, 200, 300], index=["a", "b", "c"])
print(s)
```

```
a    100
b    200
c    300
dtype: int64
```

📝 Hinglish: Apna custom index de diya

📊 3. Create Series from Dictionary

In [7]:

```
data = {"Math":90, "Science": 85, "English": 88}
s = pd.Series(data)
print(s)
```

```
Math       90
Science    85
```

```
English    88
dtype: int64
```

📝 Hinglish: Dictionary ke keys index ban gaye aur values Series me aa gayi

⚽ 4. Access Elements by Index

In [8]:

```
s = pd.Series([10, 20, 30, 40, 50], index=["a","b","c","d","e"])
print(s["c"])      # by label
print(s[2])        # by position
```

```
30
30
C:\Users\gkuir\AppData\Local\Temp\ipykernel_14552\4287591849.py:3: FutureWarning: Serie
s.__getitem__ treating keys as positions is deprecated. In a future version, integer key
s will always be treated as labels (consistent with DataFrame behavior). To access a val
ue by position, use `ser.iloc[pos]`
  print(s[2])        # by position
```

📝 Hinglish: Index se element access kiya → `"c" → 30` aur `2 → 30.`

💰 5. Real-Life Example: Student Scores

In [3]:

```
import pandas as pd
scores = pd.Series([56, 78, 89, 45, 67], index=["A","B","C","D","E"])
print(scores)
print(scores["C"])
```

```
A    56
B    78
C    89
D    45
E    67
dtype: int64
89
```

📝 Hinglish: Student ke naam → index aur unke marks values ban gaye.

📌 Bonus: NumPy Compatibility

In [5]:

```
import numpy as np

arr = np.array([1, 2, 3, 4])
s = pd.Series(arr)
print(s.values)   # NumPy array milega
print(s.index)    # Index info
```

```
[1 2 3 4]
RangeIndex(start=0, stop=4, step=1)
```

📝 Hinglish: Series ko easily NumPy array me convert kar sakte ho.

✅ What is a Pandas DataFrame (2D)?

📖 English:

In **Pandas**, a **DataFrame** is a **two-dimensional labeled data structure** with rows and columns.

- It is like a **table in Excel** or a **SQL database**, where **rows = records** and **columns = fields**.

🗣️ Hinglish:

Pandas me **DataFrame** ek **2D labeled data structure** hota hai jisme **rows aur columns** hote hain.

- Ye **Excel ki table** ya **SQL ki table** jaisa lagta hai.

🔹 Syntax:

```
import pandas as pd

pd.DataFrame(data=None, index=None, columns=None,
dtype=None, copy=False)
```

- **data** → dict, list of lists, NumPy array, Series, etc.
- **index** → row labels (optional)
- **columns** → column labels (optional)
- **dtype** → data type (optional)
- **Returns** → Pandas DataFrame object

📌 Real-Life Examples of Pandas DataFrame

🍎 1. Create DataFrame from Dictionary

In [8]:

```
import pandas as pd

data = {"Name": ["Amit", "Ravi", "Neha"], "Marks": [85, 90, 95]}
df = pd.DataFrame(data)
print(df)
```

```
   Name  Marks
0  Amit     85
1  Ravi     90
2  Neha     95
```

📝 Hinglish: Dictionary ke keys column ban gaye aur values row-wise aa gayi

🚗 2. Create DataFrame from List of Lists

In [9]:

```
data = [[1, "A"], [2, "B"], [3, "C"]]
df = pd.DataFrame(data, columns=["ID", "Grade"])
print(df)
```

```
   ID Grade
0   1     A
1   2     B
2   3     C
```

📝 Hinglish: Har list ek row ban gayi aur columns ko naam diya

📊 3. Create DataFrame with Custom Index

In [11]:

```
data = {"Math": [90, 80, 85], "Science": [88, 92, 95]}
df = pd.DataFrame(data, index=["Student1","Student2","Student3"])
print(df)
```

```
          Math  Science
Student1    90       88
Student2    80       92
Student3    85       95
```

📝 Hinglish: Apne custom row labels diye

⚽ 4. Access Rows and Columns

In [13]:

```
print(df["Math"])           # Access column
print(df.loc["Student1"])  # Access row by label
print(df.iloc[0])           # Access row by position
```

```
Student1    90
Student2    80
Student3    85
Name: Math, dtype: int64
Math       90
Science    88
Name: Student1, dtype: int64
Math       90
Science    88
Name: Student1, dtype: int64
```

📝 Hinglish: Column ko naam se aur row ko index/position se access kar sakte ho.

💰 5. Real-Life Example: Employee Data

In [15]:

```
data = {
    "Employee": ["Raj", "Simran", "Aman"],
    "Salary": [50000, 60000, 70000],
    "Dept": ["IT", "HR", "Finance"]
}
df = pd.DataFrame(data)
print(df)
```

```
  Employee  Salary     Dept
0      Raj   50000       IT
```

```
1   Simran   60000        HR
2     Aman   70000   Finance
```

📝 Hinglish: Employees ka data table jaisa DataFrame me store kar diya.

📌 **Bonus: NumPy Integration**

In [17]:

```python
import numpy as np

arr = np.arange(9).reshape(3,3)
df = pd.DataFrame(arr, columns=["A", "B", "C"])
print(df)
```

```
   A  B  C
0  0  1  2
1  3  4  5
2  6  7  8
```

📝 Hinglish: NumPy array ko directly DataFrame me convert kar liya

2. 📂 Data Input / Output (I/O Operations)

Pandas provides powerful and flexible functions to handle **data import and export** across multiple file formats.

- **CSV Files:** `read_csv()`, `to_csv()` → For reading and writing Comma-Separated Values files.
- **Excel Files:** `read_excel()`, `to_excel()` → For handling Excel spreadsheets.
- **SQL Databases:** `read_sql()` → Directly read data from SQL queries or database connections.
- **Other Formats:** Support for JSON, HTML, Parquet, and many more specialized formats.

Format	Function	Syntax	Example	Notes
CSV	Read CSV	`pd.read_csv("file.csv")`	`students = pd.read_csv("students.csv")`	CSV = Comma-Separated Values. Use `index=False` to avoid saving index.
CSV	Write CSV	`df.to_csv("output.csv", index=False)`	`students.to_csv("students_copy.csv", index=False)`	By default index is saved; `index=False` recommended.

Format	Function	Syntax	Example	Notes
Excel	Read Excel	`pd.read_excel("file.xlsx", sheet_name="Sheet1")`	`marks = pd.read_excel("marks.xlsx")`	Requires `openpyxl`. Sheet name optional.
Excel	Write Excel	`df.to_excel("output.xlsx", sheet_name="Report", index=False)`	`marks.to_excel("marks_report.xlsx", index=False)`	Sheet name optional (default "Sheet1").
SQL	Read SQL	`pd.read_sql("SELECT * FROM table", conn)`	`students = pd.read_sql("SELECT * FROM students", conn)`	`conn` = SQL connection object (sqlite3 / SQLAlchemy)
JSON	Read JSON	`pd.read_json("file.json")`	`df = pd.read_json("data.json")`	JSON = JavaScript Object Notation, structured data ke liye.
JSON	Write JSON	`df.to_json("output.json", orient="records")`	`df.to_json("output.json", orient="records")`	`orient="records"` best for row-wise JSON.
HTML	Read HTML	`pd.read_html("https://example.com")`	`df_list = pd.read_html("https://example.com/tablepage")`	Returns list of DataFrames if multiple tables exist.
HTML	Write HTML	`df.to_html("output.html", index=False)`	`df.to_html("output.html", index=False)`	Creates HTML table from DataFrame.
Parquet	Read Parquet	`pd.read_parquet("file.parquet")`	`df = pd.read_parquet("data.parquet")`	Columnar storage, fast for large datasets.
Parquet	Write Parquet	`df.to_parquet("output.parquet", index=False)`	`df.to_parquet("output.parquet", index=False)`	Efficient storage & fast read/write.

3. 🔍 Data Inspection – Explore Your Dataset (Attributes)

Function	Definition	Syntax
head()	Shows first 5 rows of dataset	`df.head()`
tail()	Shows last 5 rows of dataset	`df.tail()`
info()	Gives summary (rows, columns, dtypes, memory)	`df.info()`
dtypes	Check column data types	`df.dtypes`
shape	Returns (rows, columns)	`df.shape`
ndim	Gives dimensions (1D/2D)	`df.ndim`
describe()	Summary statistics (mean, std, etc.)	`df.describe()`
isnull()	Check missing values (True/False)	`df.isnull()`
notnull()	Check non-missing values	`df.notnull()`
sum() with isnull()	Count total missing values	`df.isnull().sum()`

📌 (i) What is df.head()?

📖 English:

In **Pandas**, the **df.head()** function returns the **first n rows of a DataFrame** (default = 5).

• It is mainly used to **quickly preview the dataset** without printing the entire thing.

🗣️ Hinglish:

Pandas me **df.head()** function **DataFrame ki pehli n rows** return karta hai (default = 5).

• Iska use **dataset ko jaldi dekhne** ke liye hota hai, bina pura data print kiye.

🔹 Syntax:

```
DataFrame.head(n=5)
```

- **n** → Number of rows to return (default = 5)
- **Returns** → Top n rows of the DataFrame

📌 Real-Life Examples of df.head()

🍎 1. Default Usage (First 5 Rows)

In [1]:

```python
import pandas as pd

data = {"Name": ["Amit", "Ravi", "Neha", "Simran", "Raj", "Aman"],
        "Marks": [85, 90, 95, 88, 76, 92]}
df = pd.DataFrame(data)

print(df.head())
```

```
     Name  Marks
0    Amit     85
1    Ravi     90
2    Neha     95
3  Simran     88
4     Raj     76
```

📝 Hinglish: Sirf pehle 5 rows dikhayega.

🚗 2. Custom Number of Rows

In [2]:

```python
print(df.head(3))
```

```
   Name  Marks
0  Amit     85
1  Ravi     90
2  Neha     95
```

📝 Hinglish: Sirf top 3 rows print hongi.

📊 3. Large Dataset Quick Check

In [19]:

```python
from IPython.display import display

df = pd.read_csv("(New) walmart Retail Row Data.csv")

display(df.head(6))
```

	City	Customer Age	Customer Name	Customer Segment	Discount	Order Date	Order ID	Order Priority	Order Quantity	Pr M
0	McKeesport	NaN	Jessica Myrick	Small Business	0.10	01/01/2012	28774	High	32	
1	Bowie	NaN	Matt Collister	Home Office	0.08	01/01/2012	13729	Not Specified	9	
2	Napa	NaN	Alan Schoenberger	Corporate	0.00	02/01/2012	37537	Low	4	
3	Montebello	NaN	Elizabeth Moffitt	Consumer	0.08	02/01/2012	44069	Critical	43	
4	Napa	NaN	Alan Schoenberger	Corporate	0.07	02/01/2012	37537	Low	43	
5	Montebello	NaN	Elizabeth Moffitt	Consumer	0.09	02/01/2012	44069	Critical	16	

6 rows × 22 columns

📝 Hinglish: Pura data load karne ki bajay sirf top 06 rows preview kar lo.

⚽ 4. Use Case in Data Cleaning

In [13]:

```
display(df.head())
print(df.info())
```

	City	Customer Age	Customer Name	Customer Segment	Discount	Order Date	Order ID	Order Priority	Order Quantity	Pr M
0	McKeesport	NaN	Jessica Myrick	Small Business	0.10	01/01/2012	28774	High	32	
1	Bowie	NaN	Matt Collister	Home Office	0.08	01/01/2012	13729	Not Specified	9	
2	Napa	NaN	Alan Schoenberger	Corporate	0.00	02/01/2012	37537	Low	4	
3	Montebello	NaN	Elizabeth Moffitt	Consumer	0.08	02/01/2012	44069	Critical	43	
4	Napa	NaN	Alan Schoenberger	Corporate	0.07	02/01/2012	37537	Low	43	

5 rows × 22 columns

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 8399 entries, 0 to 8398
Data columns (total 22 columns):
 #   Column                Non-Null Count  Dtype  
---  ------                --------------  -----  
 0   City                  8399 non-null   object 
 1   Customer Age          7496 non-null   float64
 2   Customer Name         8399 non-null   object 
 3   Customer Segment      8399 non-null   object 
 4   Discount               8399 non-null   float64
 5   Order Date             8399 non-null   object
 6   Order ID               8399 non-null   int64
 7   Order Priority         8399 non-null   object
 8   Order Quantity         8399 non-null   int64
 9   Product Base Margin    8336 non-null   float64
 10  Product Category       8399 non-null   object
 11  Product Container      8399 non-null   object
 12  Product Sub-Category   8399 non-null   object
 13  Profit                 8399 non-null   float64
 14  Region                 8399 non-null   object
 15  Row ID                 8399 non-null   int64
 16  Sales                  8399 non-null   float64
 17  Ship Date              8399 non-null   object
 18  Ship Mode              8399 non-null   object
 19  Shipping Cost          8399 non-null   float64
 20  State                  8399 non-null   object
 21  Unit Price             8399 non-null   float64
dtypes: float64(7), int64(3), object(12)
memory usage: 1.4+ MB
None
```

📝 Hinglish: Pehle head() se dekh lo data sahi load hua ya nahi, fir info() se column types check karo.

💰 5. Real-Life Example: Employee Data

In [14]:

```
data = {
    "Employee": ["Raj", "Simran", "Aman", "Neha", "Ravi", "Amit"],
    "Salary": [50000, 60000, 70000, 80000, 55000, 65000],
    "Dept": ["IT", "HR", "Finance", "IT", "HR", "Finance"]
}
df = pd.DataFrame(data)

print(df.head(4))
```

```
  Employee  Salary     Dept
0      Raj   50000       IT
1   Simran   60000       HR
2     Aman   70000  Finance
3     Neha   80000       IT
```

📝 Hinglish: Employee data me se sirf top 4 records show kar diye.

📌 (ii) What is df.tail()?

📖 English:

In **Pandas**, the **df.tail()** function returns the **last n rows of a DataFrame** (default = 5).

- It is mainly used to **peek at the bottom of the dataset**, especially when the file is large.

🗣️ Hinglish:

Pandas me **df.tail()** function **DataFrame ki last n rows** return karta hai (default = 5).

- Iska use tab hota hai jab hume **dataset ka end check** karna ho bina pura data print kiye.

🔹 Syntax:

```
DataFrame.tail(n=5)
```

- **n** → Number of rows to return (default = 5)
- **Returns** → Last n rows of the DataFrame

📌 Real-Life Examples of df.tail()

🍎 1. Default Usage (Last 5 Rows)

In [17]:

```python
import pandas as pd

data = {"Name": ["Amit", "Ravi", "Neha", "Simran", "Raj", "Aman"],
        "Marks": [85, 90, 95, 88, 76, 92]}

df = pd.DataFrame(data)

print(df.tail())
```

```
     Name  Marks
1    Ravi     90
2    Neha     95
3  Simran     88
4     Raj     76
5    Aman     92
```

📝 Hinglish: Ye last ki 5 rows dikhayega.

🚗 2. Custom Number of Rows

In [18]:

```python
print(df.tail(3))
```

```
     Name  Marks
3  Simran     88
4     Raj     76
5    Aman     92
```

📝 Hinglish: Sirf last 3 rows print hongi.

📊 3. Large Dataset Quick Check

In [20]:

```python
from IPython.display import display

df = pd.read_csv("(New) walmart Retail Row Data.csv")
```

```
display(df.tail(6))
```

	City	Customer Age	Customer Name	Customer Segment	Discount	Order Date	Order ID	Order Priority	Order Quantity
8393	Charlottesville	95.0	Jim Epp	Small Business	0.08	30/12/2015	47815	Not Specified	45
8394	Fairfield	95.0	Tony Molinari	Corporate	0.10	30/12/2015	50950	Not Specified	35
8395	Harker Heights	95.0	Matt Hagelstein	Home Office	0.09	30/12/2015	25542	Low	37
8396	Riverview	95.0	Theresa Swint	Consumer	0.10	30/12/2015	45127	Medium	10
8397	Nicholasville	95.0	Maribeth Yedwab	Home Office	0.09	30/12/2015	49344	Low	1
8398	Nicholasville	95.0	Maribeth Yedwab	Home Office	0.00	30/12/2015	49344	Low	31

6 rows × 22 columns

⚽ 4. Use Case in Data Cleaning

In [23]:

```
display(df.tail())
display(df.info())
```

	City	Customer Age	Customer Name	Customer Segment	Discount	Order Date	Order ID	Order Priority	Order Quantity	P I
8394	Fairfield	95.0	Tony Molinari	Corporate	0.10	30/12/2015	50950	Not Specified	35	
8395	Harker Heights	95.0	Matt Hagelstein	Home Office	0.09	30/12/2015	25542	Low	37	
8396	Riverview	95.0	Theresa Swint	Consumer	0.10	30/12/2015	45127	Medium	10	
8397	Nicholasville	95.0	Maribeth Yedwab	Home Office	0.09	30/12/2015	49344	Low	1	
8398	Nicholasville	95.0	Maribeth Yedwab	Home Office	0.00	30/12/2015	49344	Low	31	

5 rows × 22 columns

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 8399 entries, 0 to 8398
Data columns (total 22 columns):
 #   Column                Non-Null Count  Dtype  
---  ------                --------------  -----  
 0   City                  8399 non-null   object 
 1   Customer Age          7496 non-null   float64
 2   Customer Name          8399 non-null   object 
 3   Customer Segment       8399 non-null   object 
 4   Discount               8399 non-null   float64
 5   Order Date             8399 non-null   object 
 6   Order ID               8399 non-null   int64  
 7   Order Priority         8399 non-null   object 
 8   Order Quantity         8399 non-null   int64  
 9   Product Base Margin    8336 non-null   float64
 10  Product Category       8399 non-null   object 
 11  Product Container      8399 non-null   object 
 12  Product Sub-Category   8399 non-null   object 
 13  Profit                 8399 non-null   float64
 14  Region                 8399 non-null   object 
 15  Row ID                 8399 non-null   int64  
 16  Sales                  8399 non-null   float64
 17  Ship Date              8399 non-null   object 
 18  Ship Mode              8399 non-null   object 
 19  Shipping Cost          8399 non-null   float64
 20  State                  8399 non-null   object 
 21  Unit Price             8399 non-null   float64
dtypes: float64(7), int64(3), object(12)
memory usage: 1.4+ MB
None
```

📝 Hinglish: Pehle tail() se end ka data dekh lo sahi load hua ya nahi, fir info() se types check karo.

💰 5. Real-Life Example: Employee Data

In [25]:

```python
data = {
    "Employee": ["Raj", "Simran", "Aman", "Neha", "Ravi", "Amit"],
    "Salary": [50000, 60000, 70000, 80000, 55000, 65000],
    "Dept": ["IT", "HR", "Finance", "IT", "HR", "Finance"]
}

df = pd.DataFrame(data)

print(df.tail(4))
```

```
  Employee  Salary     Dept
2     Aman   70000  Finance
3     Neha   80000       IT
4     Ravi   55000       HR
5     Amit   65000  Finance
```

📝 Hinglish: Employee data me se sirf last 4 records show kar diye.

📌 (iii) What is df.info()?

📖 English:

In **Pandas**, the **df.info()** function provides a **concise summary of the DataFrame**.

- It shows **number of rows**, **column names**, **non-null values**, and **data types** of each column.
- Basically, it’s like a quick **health report** of your dataset.

🗣️ Hinglish:

Pandas me **df.info()** function ek **quick summary** deta hai DataFrame ka.

- Isme **rows**, **columns**, **non-null values** aur **har column ka data type** dikhata hai.
- Bhai simple bolun toh — dataset ka **status report** milta hai ek line me.

🔹 Syntax:

```
DataFrame.info(verbose=None, memory_usage=None)
```

- **verbose** → Detailed info chahiye ya nahi (default = None)
- **memory_usage** → Memory consumption bhi dikhana hai ya nahi (default = True)
- **Returns** → Prints DataFrame summary (no return object)

📌 Real-Life Examples of df.info()

🍎 1. Basic Usage

In [26]:

```python
import pandas as pd

data = {"Name": ["Amit", "Ravi", "Neha", "Simran", "Raj"],
        "Marks": [85, 90, 95, 88, 76]}

df = pd.DataFrame(data)

print(df.info())
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 5 entries, 0 to 4
Data columns (total 2 columns):
 #   Column  Non-Null Count  Dtype 
---  ------  --------------  ----- 
 0   Name    5 non-null      object
 1   Marks   5 non-null      int64 
dtypes: int64(1), object(1)
memory usage: 212.0+ bytes
None
```

📝 Hinglish: Ye rows, columns aur data types ka quick summary dikhayega.

🚗 2. With Missing Values

In [27]:

```python
data = {"Name": ["Amit", "Ravi", "Neha", None, "Raj"],
        "Marks": [85, 90, None, 88, 76]}

df = pd.DataFrame(data)
```

```
print(df.info())
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 5 entries, 0 to 4
Data columns (total 2 columns):
 #   Column  Non-Null Count  Dtype  
---  ------  --------------  -----  
 0   Name    4 non-null      object 
 1   Marks   4 non-null      float64
dtypes: float64(1), object(1)
memory usage: 212.0+ bytes
None
```

📝 Hinglish: Non-null values kam hongi, matlab missing data hai.

📊 3. Large Dataset Check

In [29]:

```
df = pd.read_csv("(New) walmart Retail Row Data.csv")
print(df.info())
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 8399 entries, 0 to 8398
Data columns (total 22 columns):
 #   Column                Non-Null Count  Dtype  
---  ------                --------------  -----  
 0   City                  8399 non-null   object 
 1   Customer Age          7496 non-null   float64
 2   Customer Name         8399 non-null   object 
 3   Customer Segment      8399 non-null   object 
 4   Discount              8399 non-null   float64
 5   Order Date            8399 non-null   object 
 6   Order ID              8399 non-null   int64  
 7   Order Priority        8399 non-null   object 
 8   Order Quantity        8399 non-null   int64  
 9   Product Base Margin   8336 non-null   float64
 10  Product Category      8399 non-null   object 
 11  Product Container     8399 non-null   object 
 12  Product Sub-Category  8399 non-null   object 
 13  Profit                8399 non-null   float64
 14  Region                8399 non-null   object 
 15  Row ID                8399 non-null   int64  
 16  Sales                 8399 non-null   float64
 17  Ship Date             8399 non-null   object 
 18  Ship Mode             8399 non-null   object 
 19  Shipping Cost         8399 non-null   float64
 20  State                 8399 non-null   object 
 21  Unit Price            8399 non-null   float64
dtypes: float64(7), int64(3), object(12)
memory usage: 1.4+ MB
None
```

💰 4. Real-Life Example: Employee Data

In [30]:

```
data = {
    "Employee": ["Raj", "Simran", "Aman", "Neha", "Ravi"],
    "Salary": [50000, 60000, 70000, None, 55000],
    "Dept": ["IT", "HR", "Finance", "IT", "HR"]
}

df = pd.DataFrame(data)

df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 5 entries, 0 to 4
Data columns (total 3 columns):
 #   Column    Non-Null Count  Dtype  
---  ------    --------------  -----  
 0   Employee  5 non-null      object 
 1   Salary    4 non-null      float64
 2   Dept      5 non-null      object 
dtypes: float64(1), object(2)
memory usage: 252.0+ bytes
```

📝 Hinglish: Salary column me ek None hoga toh Non-Null Count kam ho jaayega.

📌 (iv) What is df.dtypes?

📖 English:

In **Pandas**, the **df.dtypes** attribute returns the **data type (dtype) of each column** in the DataFrame.

- It helps to know whether a column is **integer**, **float**, **string (object)**, or **datetime**, etc.
- Basically, it's a quick way to **check the format of your columns**.

🗣️ Hinglish:

Pandas me **df.dtypes** attribute **har column ka data type** batata hai.

- Matlab ye column **int** hai, **float** hai, **object (string)** hai ya **datetime** hai, sab ekdum seedhe point pe dikhata hai.
- Bhai simple bolun toh — **columns ka format check karne ka shortcut** hai.

🔹 Syntax:

```
DataFrame.dtypes
```

- **Parameters** → None (it's an attribute, not a function)
- **Returns** → A Pandas Series with column names as index and their data types as values

📌 Real-Life Examples of df.dtypes

🍎 1. Basic Usage

In [31]:

```
import pandas as pd

data = {"Name": ["Amit", "Ravi", "Neha"],
        "Marks": [85, 90, 95],
        "Passed": [True, True, False]}

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

```
Name      object
Marks      int64
Passed      bool
dtype: object
```

📝 Hinglish: Ye batayega ki `Name` = object, `Marks` = int, `Passed` = bool.

🚗 2. With Mixed Types

In [32]:

```
data = {"ID": [1, 2, 3],
        "Salary": [50000.5, 60000.0, 70000.25],
        "JoinDate": pd.to_datetime(["2020-01-01", "2021-02-15", "2022-03-20"])}

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

```
ID                   int64
Salary             float64
JoinDate    datetime64[ns]
dtype: object
```

📝 Hinglish: ID = int64, Salary = float64, JoinDate = datetime64[ns].

📊 3. Large Dataset Quick Check

In [33]:

```
from IPython.display import display

df = pd.read_csv("(New) walmart Retail Row Data.csv")

display(df.head(6))
```

	City	Customer Age	Customer Name	Customer Segment	Discount	Order Date	Order ID	Order Priority	Order Quantity	Pr M
0	McKeesport	NaN	Jessica Myrick	Small Business	0.10	01/01/2012	28774	High	32	
1	Bowie	NaN	Matt Collister	Home Office	0.08	01/01/2012	13729	Not Specified	9	
2	Napa	NaN	Alan Schoenberger	Corporate	0.00	02/01/2012	37537	Low	4	
3	Montebello	NaN	Elizabeth Moffitt	Consumer	0.08	02/01/2012	44069	Critical	43	

	City	Customer Age	Customer Name	Customer Segment	Discount	Order Date	Order ID	Order Priority	Order Quantity	Pr M
4	Napa	NaN	Alan Schoenberger	Corporate	0.07	02/01/2012	37537	Low	43	
5	Montebello	NaN	Elizabeth Moffitt	Consumer	0.09	02/01/2012	44069	Critical	16	

6 rows × 22 columns

⚽ 4. Use Case in Data Cleaning

In [37]:

```
print(df["Sales"].dtypes)
df["Sales"]=df["Sales"].astype(int)
print(df["Sales"].dtypes)
```

```
float64
int64
```

📝 Hinglish: Pehle dtypes check karo, fir column ko correct type me convert karo.

💰 5. Real-Life Example: Employee Data

In [38]:

```
data = {
    "Employee": ["Raj", "Simran", "Aman"],
    "Salary": [50000, 60000, 70000],
    "Dept": ["IT", "HR", "Finance"]
}

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

```
Employee    object
Salary       int64
Dept        object
dtype: object
```

📝 Hinglish: Employee aur Dept string (object) honge, Salary integer.

📌 (v) What is df.shape?

📖 English:

In **Pandas**, the **df.shape** attribute returns the **dimensions of the DataFrame** in the form of a tuple **(rows, columns)**.

- It's super handy to quickly **check how big your dataset is**.

🗣️ Hinglish:

Pandas me **df.shape** ek tuple return karta hai jisme **(rows, columns)** hota hai.

• Bhai simple bole toh — kitni **rows** aur kitne **columns** hai dataset me, ye ekdum quick check mil jaata hai.

🔹 Syntax:

```
DataFrame.shape
```

• **Parameters** → None (it’s an attribute)

• **Returns** → Tuple of (*number_of_rows*, *number_of_columns*)

📌 Real-Life Examples of df.shape

🍎 1. Basic Usage

In [40]:

```
import pandas as pd

data = {"Name": ["Amit", "Ravi", "Neha"],
        "Marks": [85, 90, 95]}

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

```
(3, 2)
```

📝 Hinglish: Output `(3, 2)` matlab 3 rows aur 2 columns.

🚗 2. Large Dataset Check

In [43]:

```
pd.set_option("display.width", None)
df = pd.read_csv("(New) walmart Retail Row Data.csv")
print(df.shape)
```

```
(8399, 22)
```

📝 Hinglish: Bina pura dataset print kiye pata chal jaayega ki dataset kitna bada hai.

📊 3. After Filtering

In [46]:

```
print(df.shape)
filtered_df = df[df["Sales"] >= 5600]
print(filtered_df.shape)
```

```
(8399, 22)
(700, 22)
```

📝 Hinglish: Filter ke baad rows kam ho gayi, shape change ho gaya.

⚽ 4. Use Case in Data Cleaning

In [47]:

```
print("Before cleaning:", df.shape)
df = df.dropna()
print("After cleaning:", df.shape)
```

```
Before cleaning: (8399, 22)
After cleaning: (7440, 22)
```

📝 Hinglish: Missing values remove karne ke baad rows kam ho gayi, shape me turant difference dikh gaya.

💰 5. Real-Life Example: Employee Data

In [48]:

```
data = {
    "Employee": ["Raj", "Simran", "Aman", "Neha"],
    "Salary": [50000, 60000, 70000, 80000],
    "Dept": ["IT", "HR", "Finance", "IT"]
}

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

```
(4, 3)
```

📝 Hinglish: Output `(4, 3)` matlab 4 employees aur 3 columns.

📌 (vi) What is df.ndim?

📖 English:

In **Pandas**, the **df.ndim** attribute returns the **number of dimensions** of your data.

- For a **Series**, **ndim = 1** (1D data).
- For a **DataFrame**, **ndim = 2** (2D data).

🗣️ Hinglish:

df.ndim batata hai ki tumhara data kitne **dimension** ka hai.

👉 **Series** ek column hota hai toh **ndim = 1**.

👉 **DataFrame** me rows aur columns dono hote hain toh **ndim = 2**.

🔹 Syntax:

```
DataFrame.ndim
Series.ndim
```

- **Parameters** → None (it's an attribute)
- **Returns** → Integer (mostly 1 ya 2)

📌 Real-Life Examples of df.ndim

🍎 1. Series Example

In [49]:

```python
import pandas as pd

s = pd.Series([10, 20, 30])
print(s.ndim)
```

```
1
```

📝 Hinglish: Output → 1 matlab Series ek 1D structure hai.

🚗 2. DataFrame Example

In [50]:

```python
data = {"Name": ["Amit", "Ravi", "Neha"],
        "Marks": [85, 90, 95]}

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

```
2
```

📝 Hinglish: Output → 2 matlab DataFrame 2D structure hai (rows + columns).

📊 3. Large Dataset Example

In [51]:

```python
pd.set_option("display.width", None)
df = pd.read_csv("(New) walmart Retail Row Data.csv")
print(df.ndim)
```

```
2
```

📝 Hinglish: Chahe dataset 1 row ho ya 1 crore rows, DataFrame hamesha 2D hota hai.

⚽ 4. Filtering ke Baad bhi

In [52]:

```python
filtered_df = df[df["Sales"] > 80]
print(filtered_df.ndim)
```

```
2
```

📝 Hinglish: Filtering se rows kam ho jaayengi, lekin dimension wahi 2D rahega.

💰 5. Real-Life Example: Employee Data

In [53]:

```python
data = {
    "Employee": ["Raj", "Simran", "Aman", "Neha"],
    "Salary": [50000, 60000, 70000, 80000],
    "Dept": ["IT", "HR", "Finance", "IT"]
}

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

```
2
```

📝 Hinglish: Output → 2, matlab employee data ek table (2D) ke form me hai.

📌 (vii) What is df.describe()?

📖 English:

In **Pandas**, the **df.describe()** function generates **summary statistics** of your dataset.

• For **numeric columns** → it shows **count, mean, std, min, quartiles (25%, 50%, 75%)**, and **max**.

• For **object columns** (when `include="object"`) → it shows **count, unique, top**, and **frequency**.

🗣️ Hinglish:

df.describe() ek shortcut hai jo tumhare data ka ek chhota **report card** banata hai.

👉 **Numeric data** ke liye — average, spread, min-max, aur quartile values deta hai.

👉 **Text data** ke liye — unique values aur sabse zyada aane wali value (mode) dikhata hai.

🔹 Syntax:

```
DataFrame.describe(percentiles=None, include=None,
exclude=None)
```

• **Parameters**:

- **percentiles** → Extra percentiles dikhane ke liye (default [0.25, 0.5, 0.75])
- **include** → Data types specify karna (like "all", ["object"], ["number"])
- **exclude** → Kaunse data types exclude karne hain

• **Returns** → DataFrame of summary stats

📌 Real-Life Examples of df.describe()

🍎 1. Basic Usage

In [54]:

```
import pandas as pd

data = {"Marks": [85, 90, 95, 88, 92]}
df = pd.DataFrame(data)

print(df.describe())
```

```
           Marks
count   5.000000
mean   90.000000
std     3.807887
min    85.000000
25%    88.000000
50%    90.000000
75%    92.000000
max    95.000000
```

📝 Hinglish: Ye `count, mean, std, min, 25%, 50%, 75%, max` dikhayega for "Marks".

🚗 2. On Multiple Columns

In [55]:

```
data = {
    "Maths": [85, 90, 95, 88, 92],
    "Science": [80, 85, 89, 91, 87]
}
df = pd.DataFrame(data)

print(df.describe())
```

```
           Maths    Science
count   5.000000   5.000000
mean   90.000000  86.400000
std     3.807887   4.219005
min    85.000000  80.000000
25%    88.000000  85.000000
50%    90.000000  87.000000
75%    92.000000  89.000000
max    95.000000  91.000000
```

📝 Hinglish: Har numeric column ka summary aa jaayega ek table me.

📊 3. Describe with Object Data

In [58]:

```
data = {"Name": ["Amit", "Ravi", "Ravi", "Neha", "Amit"]}
df = pd.DataFrame(data)

print(df.describe(include="object"))
```

```
        Name
count      5
unique     3
top     Amit
freq       2
```

📝 Hinglish: Output → count (5), unique (3), top ("Amit"), freq (2).

⚽ 4. Custom Percentiles

In [59]:

```
df = pd.DataFrame({"Marks": [50, 60, 70, 80, 90, 100]})
print(df.describe(percentiles=[0.1, 0.5, 0.9]))
```

```
            Marks
count    6.000000
mean    75.000000
std     18.708287
min     50.000000
10%     55.000000
50%     75.000000
90%     95.000000
max    100.000000
```

📝 Hinglish: `10%, 50%, 90%` wale percentiles bhi show honge.

💰 5. Real-Life Example: Employee Salaries

In [60]:

```
data = {
    "Employee": ["Raj", "Simran", "Aman", "Neha", "Raj"],
    "Salary": [50000, 60000, 70000, 80000, 50000]
}

df = pd.DataFrame(data)
print(df.describe())
print(df.describe(include="object"))
```

```
            Salary
count      5.00000
mean   62000.00000
std    13038.40481
min    50000.00000
25%    50000.00000
50%    60000.00000
75%    70000.00000
max    80000.00000
       Employee
count         5
unique        4
top         Raj
freq          2
```

📝 Hinglish: Salary ka mean, min, max milega. Employee column ka unique aur top value bhi aa jaayega.

📌 (viii) What is df.isnull()?

📖 English:

In **Pandas**, the **df.isnull()** function checks for **missing values (NaN)** in a DataFrame.

- It returns a DataFrame of the same shape but with **Boolean values**.
- **True** → if the value is missing (**NaN**).
- **False** → if the value is present.

🗣️ Hinglish:

df.isnull() ek magnifying glass ki tarah hai jo dataset me **missing values** dhoondhta hai.

👉 Agar value missing hai toh **True** dikhayega.

👉 Agar value present hai toh **False** dikhayega.

🔹 Syntax:

```
DataFrame.isnull()
```

- **Parameters** → None (simple function)
- **Returns** → DataFrame of Boolean values (**True/False**)

📌 Real-Life Examples of df.isnull()

🍎 1. Basic Usage

In [1]:

```
import pandas as pd

data = {"Name": ["Amit", "Ravi", None],
        "Marks": [85, None, 95]}

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

```
    Name  Marks
0  False  False
1  False   True
2   True  False
```

📝 Hinglish: Jahan value missing hai, wahan True aa jaayega.

🚗 2. Checking with sum()

In [2]:

```
print(df.isnull().sum())
```

```
Name     1
Marks    1
```

```
dtype: int64
```

📝 Hinglish: Har column me kitni missing values hain, count me de dega.

📊 3. On Large Dataset

In [3]:

```python
df = pd.read_csv("(New) walmart Retail Row Data.csv")
print(df.isnull().sum())
```

```
City                      0
Customer Age            903
Customer Name             0
Customer Segment          0
Discount                  0
Order Date                0
Order ID                  0
Order Priority            0
Order Quantity            0
Product Base Margin      63
Product Category          0
Product Container         0
Product Sub-Category      0
Profit                    0
Region                    0
Row ID                    0
Sales                     0
Ship Date                 0
Ship Mode                 0
Shipping Cost             0
State                     0
Unit Price                0
dtype: int64
```

📝 Hinglish: Jaldi se pata chal jaayega kaunse columns me kitna data missing hai.

⚽ 4. Use Case in Data Cleaning

In [4]:

```python
print("Before cleaning:\n", df.isnull().sum())
df = df.dropna()
print("After cleaning:\n", df.isnull().sum())
```

```
Before cleaning:
 City                      0
Customer Age            903
Customer Name             0
Customer Segment          0
Discount                  0
Order Date                0
Order ID                  0
Order Priority            0
Order Quantity            0
Product Base Margin      63
Product Category          0
Product Container         0
Product Sub-Category      0
Profit                    0
Region                    0
```

```
Row ID                    0
Sales                     0
Ship Date                 0
Ship Mode                 0
Shipping Cost             0
State                     0
Unit Price                0
dtype: int64
After cleaning:
 City                    0
Customer Age            0
Customer Name           0
Customer Segment        0
Discount                0
Order Date              0
Order ID                0
Order Priority          0
Order Quantity          0
Product Base Margin     0
Product Category        0
Product Container       0
Product Sub-Category    0
Profit                  0
Region                  0
Row ID                  0
Sales                   0
Ship Date               0
Ship Mode               0
Shipping Cost           0
State                   0
Unit Price              0
dtype: int64
```

📝 Hinglish: Dropna ke baad missing values zero ho jaati hain.

💰 5. Real-Life Example: Employee Data

In [5]:

```
data = {
    "Employee": ["Raj", "Simran", "Aman", None],
    "Salary": [50000, None, 70000, 80000]
}

df = pd.DataFrame(data)
print(df.isnull())
print(df.isnull().sum())
```

```
   Employee  Salary
0     False   False
1     False    True
2     False   False
3      True   False
Employee    1
Salary      1
dtype: int64
```

📝 Hinglish: Salary aur Employee column me missing data turant pakad me aa jaata hai.

📌 (ix) What is df.notnull()?

📖 English:

In **Pandas**, the **df.notnull()** function checks for **non-missing values** in a DataFrame.

- It returns a DataFrame of the same shape but with **Boolean values**.
- **True** → if the value is present (not NaN).
- **False** → if the value is missing (NaN).

🗣️ Hinglish:

df.notnull() ek filter ki tarah hai jo batata hai dataset me kaunse **values available** hain.

👉 Agar value available hai toh **True**.

👉 Agar value missing hai toh **False**.

🔹 Syntax:

```
DataFrame.notnull()
```

- **Parameters** → None (simple function)
- **Returns** → DataFrame of Boolean values (**True/False**)

📌 Real-Life Examples of df.notnull()

🍎 1. Basic Usage

In [6]:

```
import pandas as pd

data = {"Name": ["Amit", "Ravi", None],
        "Marks": [85, None, 95]}

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

```
    Name  Marks
0   True   True
1   True  False
2  False   True
```

📝 Hinglish: Jahan value present hai, wahan True aa jaayega.

🚗 2. Checking with sum()

In [7]:

```
print(df.notnull().sum())
```

```
Name     2
Marks    2
dtype: int64
```

📝 Hinglish: Har column me kitni values available hain, count me de dega.

📊 3. On Large Dataset

In [8]:

```
df = pd.read_csv("(New) walmart Retail Row Data.csv")
print(df.notnull().sum())
```

```
City                    8399
Customer Age            7496
Customer Name           8399
Customer Segment        8399
Discount                8399
Order Date              8399
Order ID                8399
Order Priority          8399
Order Quantity          8399
Product Base Margin     8336
Product Category        8399
Product Container       8399
Product Sub-Category    8399
Profit                  8399
Region                  8399
Row ID                  8399
Sales                   8399
Ship Date               8399
Ship Mode               8399
Shipping Cost           8399
State                   8399
Unit Price              8399
dtype: int64
```

📝 Hinglish: Bada dataset load karke check kar sakte ho ki kitna data actually available hai.

⚽ 4. Use Case in Data Cleaning

In [9]:

```
print("Available values before cleaning:\n", df.notnull().sum())
df = df.fillna("Unknown")
print("Available values after cleaning:\n", df.notnull().sum())
```

```
Available values before cleaning:
 City                    8399
Customer Age            7496
Customer Name           8399
Customer Segment        8399
Discount                8399
Order Date              8399
Order ID                8399
Order Priority          8399
Order Quantity          8399
Product Base Margin     8336
Product Category        8399
Product Container       8399
Product Sub-Category    8399
Profit                  8399
Region                  8399
Row ID                  8399
Sales                   8399
Ship Date               8399
Ship Mode               8399
Shipping Cost           8399
State                   8399
Unit Price              8399
dtype: int64
Available values after cleaning:
 City                   8399
Customer Age            8399
Customer Name           8399
Customer Segment        8399
Discount                8399
Order Date              8399
Order ID                8399
Order Priority          8399
Order Quantity          8399
Product Base Margin     8399
Product Category        8399
Product Container       8399
Product Sub-Category    8399
Profit                  8399
Region                  8399
Row ID                  8399
Sales                   8399
Ship Date               8399
Ship Mode               8399
Shipping Cost           8399
State                   8399
Unit Price              8399
dtype: int64
```

📝 Hinglish: Fillna lagane ke baad sab values True ho jaati hain (matlab missing hat gaya).

💰 5. Real-Life Example: Employee Data

In [10]:

```
data = {
    "Employee": ["Raj", "Simran", "Aman", None],
    "Salary": [50000, None, 70000, 80000]
}

df = pd.DataFrame(data)
print(df.notnull())
print(df.notnull().sum())
```

```
   Employee  Salary
0      True    True
1      True   False
2      True    True
```

```
3      False    True
Employee     3
Salary       3
dtype: int64
```

📝 **Hinglish: Employee aur Salary column me sirf wahi rows True hain jahan data available hai.**

4. 🎯 Data Selection & Indexing

Operation	Definition	Syntax / Example

Select Columns	Select single or multiple columns.	`df["col"]`
		`df[["col1","col2"]]`

Select Rows (loc)	Select rows by labels (index names).	`df.loc[0]`
		`df.loc[0:5]`

Select Rows (iloc)	Select rows by integer index position.	`df.iloc[0]`
		`df.iloc[0:5, 1:3]`

Conditional Selection	Filter rows based on condition(s).	`df[df["age"] > 25]`
		`df[(df["age"] > 25) & (df["city"]=="Delhi")]`

Slicing	Slice rows and columns like lists/arrays.	`df[0:5]`
		`df.iloc[:, 0:2]`

Set Index	Set a column as index.	`df.set_index("id")`
Reset Index	Reset index back to default 0..n.	`df.reset_index()`

📌 (i) What is Column Selection?

📏 **Definition — Extract Specific Data from DataFrame**

📖 English:

In Pandas, **Column Selection** means extracting one or more specific columns from a DataFrame.

- **Single Column** → Returns a **Pandas Series**.
- **Multiple Columns** → Returns a **Pandas DataFrame.**

🗣️ Hinglish:

Pandas me **column select** karna matlab apne DataFrame se sirf wohi data nikalna jo tumhe chahiye.
👉 Ek column select karoge toh **Series** milega.
👉 Do ya zyada columns select karoge toh **DataFrame** milega.

🔹 Syntax:

```
Select single column
df["col_name"]


Select multiple columns
df[["col1", "col2"]]
```

- **Parameters** → Column name(s) (string ya list of strings)
- **Returns** → Series (single col) / DataFrame (multiple cols)

📌 Real-Life Examples of Column Selection

🍎 1. Single Column Selection

In [7]:

```python
import pandas as pd

data = {"Name": ["Amit", "Ravi", "Simran"],
        "Marks": [85, 90, 95]}

df = pd.DataFrame(data)
print(df['Name'].to_frame())   # to_frame() function show heading name in series
```

```
     Name
0    Amit
1    Ravi
2  Simran
```

📝 Hinglish: Sirf **Name** column nikala, output ek **Series** hoga.

🚗 2. Multiple Column Selection

In [9]:

```python
print(df[["Name","Marks"]])
```

```
     Name  Marks
0    Amit     85
1    Ravi     90
2  Simran     95
```

📝 Hinglish: Yahan dono columns ek saath nikal liye, output ek **DataFrame** hoga.

📊 3. With Large Dataset

In [28]:

```python
import pandas as pd
from IPython.display import display
df = pd.read_csv("(New) walmart Retail Row Data.csv")

display(df.head(3))
display(df[["Customer Name","Customer Age","Discount","Order Quantity","Profit"]].tail()
```

	City	Customer Age	Customer Name	Customer Segment	Discount	Order Date	Order ID	Order Priority	Order Quantity	Pr M
0	McKeesport	NaN	Jessica Myrick	Small Business	0.10	01/01/2012	28774	High	32	
1	Bowie	NaN	Matt Collister	Home Office	0.08	01/01/2012	13729	Not Specified	9	
2	Napa	NaN	Alan Schoenberger	Corporate	0.00	02/01/2012	37537	Low	4	

	Customer Name	Customer Age	Discount	Order Quantity	Profit
8394	Tony Molinari	95.0	0.10	35	-15.07
8395	Matt Hagelstein	95.0	0.09	37	-18.66
8396	Theresa Swint	95.0	0.10	10	-1.29
8397	Maribeth Yedwab	95.0	0.09	1	-745.20
8398	Maribeth Yedwab	95.0	0.00	31	27.85

📝 Hinglish: Bada dataset me sirf relevant columns nikal ke analysis easy ho jata hai.

⚽ 4. Column Selection + Operations

In [34]:

```python
print(round(df["Discount"].mean()*100,2), "%")
```

```
4.97 %
```

📝 Hinglish: Direct ek column select karke uska average nikal liya.

💰 5. Real-Life Example: Employee Data

In [38]:

```python
data = {
    "Employee": ["Raj", "Simran", "Aman"],
    "Salary": [50000, 60000, 70000],
    "Department": ["HR", "IT", "Finance"]
}

df = pd.DataFrame(data)
print(df["Salary"].to_frame())           # Single column
print("\n",df[["Employee", "Salary"]])  # Multiple columns
```

```
   Salary
0   50000
1   60000
2   70000

   Employee  Salary
0       Raj   50000
1    Simran   60000
2      Aman   70000
```

📝 Hinglish: Salary alag nikal sakte ho ya Employee + Salary dono ek saath dekh sakte ho.

📌 (ii) What is Row Selection (using loc)?

📏 Definition — Extract Specific Rows from DataFrame

📖 English:

Row selection in **Pandas** using **.loc** means choosing specific rows based on labels (index names). It allows you to fetch data by index values instead of positions.

• **Single Row** → Returns a **Pandas Series**.

• **Multiple Rows** → Returns a **Pandas DataFrame**.

🗣️ Hinglish:

Pandas me **loc** use karke row select karna matlab tum index ke **label** ke hisaab se rows nikal sakte ho.

👉 Ek row select karoge toh **Series** milega.

👉 Zyada rows select karoge toh **DataFrame** milega.

🔹 Syntax:

```
Select single row by label
df.loc[label]


Select multiple rows
df.loc[start_label:end_label]
```

```
Select rows + specific columns
df.loc[start_label:end_label, ["col1", "col2"]]
```

- **Parameters**:

  👉 **labels** → index values or range

  👉 **columns** → (optional) specify selected columns

- **Returns** → Series (single row) / DataFrame (multiple rows)

📌 Real-Life Examples of Row Selection (using loc)

🍎 1. Single Row Selection

In [40]:

```python
import pandas as pd

data = {"Name": ["Amit", "Ravi", "Simran"],
        "Marks": [85, 90, 95]}

df = pd.DataFrame(data)
print(df.loc[0])
```

```
Name      Amit
Marks       85
Name: 0, dtype: object
```

📝 Hinglish: Index **0** wali row nikal li, output ek **Series** hoga.

🚗 2. Multiple Row Selection

In [42]:

```python
print(df.loc[0:1])
```

```
   Name  Marks
0  Amit     85
1  Ravi     90
```

📝 Hinglish: Yahan 0 se 1 tak ke rows nikal liye, output ek **DataFrame** hoga.

📊 3. With Large Dataset

In [48]:

```python
from IPython.display import display
import pandas as pd

df = pd.read_csv("(New) walmart Retail Row Data.csv")
display(df.head(5))

display(df.loc[2:4, ["Customer Name","City","Customer Age"]])
```

	City	Customer Age	Customer Name	Customer Segment	Discount	Order Date	Order ID	Order Priority	Order Quantity	Pr M
0	McKeesport	NaN	Jessica Myrick	Small Business	0.10	01/01/2012	28774	High	32	
1	Bowie	NaN	Matt Collister	Home Office	0.08	01/01/2012	13729	Not Specified	9	
2	Napa	NaN	Alan Schoenberger	Corporate	0.00	02/01/2012	37537	Low	4	
3	Montebello	NaN	Elizabeth Moffitt	Consumer	0.08	02/01/2012	44069	Critical	43	
4	Napa	NaN	Alan Schoenberger	Corporate	0.07	02/01/2012	37537	Low	43	

	Customer Name	City	Customer Age
2	Alan Schoenberger	Napa	NaN
3	Elizabeth Moffitt	Montebello	NaN
4	Alan Schoenberger	Napa	NaN

📝 Hinglish: Bade dataset me sirf pehli 6 rows ke Customer Name, City aur Customer Age nikal liye.

⚽ 4. Row Selection with Conditions

In [64]:

```
display(df.loc[df["Order Quantity"] > 45, ["Order Quantity"]].count())
display(df.loc[df["Order Quantity"] > 45])
```

```
Order Quantity    849
dtype: int64
```

	City	Customer Age	Customer Name	Customer Segment	Discount	Order Date	Order ID	Order Priority	Order Quantity	Pro E Ma
17	Baton Rouge	NaN	Andy Reiter	Corporate	0.05	03/01/2012	9637	Low	49	
21	Bedford	NaN	Darren Budd	Consumer	0.07	05/01/2012	57253	Critical	48	
24	Bedford	NaN	Darren Budd	Consumer	0.08	05/01/2012	57253	Critical	49	
44	Hilton Head Island	NaN	Craig Leslie	Consumer	0.01	06/01/2012	14274	Not Specified	46	
46	Fairfax	NaN	Maribeth Schnelling	Corporate	0.09	06/01/2012	41094	Low	46	
...	...	...	...	...	...	...	...	...	...	
8337	Red Wing	79.0	Ken Heidel	Corporate	0.05	16/12/2015	33570	Not Specified	46	

	City	Customer Age	Customer Name	Customer Segment	Discount	Order Date	Order ID	Order Priority	Order Quantity	Pro E Ma
8341	Bountiful	81.0	Harold Pawlan	Small Business	0.00	18/12/2015	19745	High	50	
8360	North Pembroke	82.0	Nora Paige	Small Business	0.06	21/12/2015	23619	Medium	48	
8371	Phoenix	86.0	Cindy Chapman	Corporate	0.10	25/12/2015	30469	Not Specified	46	
8391	Horn Lake	88.0	Jennifer Jackson	Home Office	0.10	29/12/2015	29216	Critical	46	

849 rows × 22 columns

📝 Hinglish: Sirf un rows ko select kar liya jahan Marks 45 se zyada hain.

💰 5. Real-Life Example: Employee Data

In [66]:

```
data = {
    "Employee": ["Raj", "Simran", "Aman", "Neha"],
    "Salary": [50000, 60000, 70000, 80000],
    "Department": ["HR", "IT", "Finance", "HR"]
}

df = pd.DataFrame(data)

# Single row
print(df.loc[2])

# Multiple rows with specific columns
print("\n", df.loc[1:3, ["Employee", "Salary"]])
```

```
Employee         Aman
Salary          70000
Department    Finance
Name: 2, dtype: object

   Employee  Salary
1    Simran   60000
2      Aman   70000
3      Neha   80000
```

📝 Hinglish: Ek row alag se nikal sakte ho, ya phir ek range ke rows sirf specific columns ke saath.

📌 (iii) What is Row Selection (using iloc)?

📏 Definition — Extract Specific Rows by Integer Position

📖 English:

Row selection in Pandas using **.iloc** means choosing specific rows based on their **integer index position (0-based)**. It works similar to Python list/array indexing.

- **Single Row** → Returns a **Pandas Series**.
- **Multiple Rows** → Returns a **Pandas DataFrame**.

🗣️ Hinglish:

.iloc ekdum Python list slicing jaisa hai.
👉 Row position ke hisaab se data nikalta hai (0 se start hota hai).
👉 Ek row select karoge toh **Series** milega.
👉 Range ya multiple rows select karoge toh **DataFrame** milega.

🔹 Syntax:

```
Select single row by integer index
df.iloc[index]


Select multiple rows (range)
df.iloc[start:end]


Select rows + specific columns
df.iloc[start:end, col_start:col_end]
```

- **Parameters**:
  👉 **index / start:end** → row numbers (integer positions)
  👉 **col_start:col_end** → (optional) column positions

- **Returns** → Series (single row) / DataFrame (multiple rows)

📌 Real-Life Examples of Row Selection with iloc

🍎 1. Single Row Selection

In [67]:

```
import pandas as pd

data = {"Name": ["Amit", "Ravi", "Simran"],
        "Marks": [85, 90, 95]}

df = pd.DataFrame(data)
print(df.iloc[0])
```

```
Name     Amit
Marks      85
Name: 0, dtype: object
```

📝 Hinglish: Position 0 wali row nikal li, output ek Series hoga.

🚗 2. Multiple Row Selection

In [68]:

```
print(df.iloc[0:2])
```

```
   Name  Marks
0  Amit     85
1  Ravi     90
```

📝 Hinglish: Yahan index 0 se 1 tak ke rows nikal liye, output ek **DataFrame** hoga.

📊 3. Selecting Rows + Specific Columns

In [70]:

```
print(df.iloc[0:2, 0:1])
```

```
   Name
0  Amit
1  Ravi
```

📝 Hinglish: Sirf first 2 rows aur pehla column select kiya.

⚽ 4. With Large Dataset

In [73]:

```
from IPython.display import display
import pandas as pd

df = pd.read_csv("(New) walmart Retail Row Data.csv")
display(df.head(3))

display(df.iloc[0:5, 2:5])
```

	City	Customer Age	Customer Name	Customer Segment	Discount	Order Date	Order ID	Order Priority	Order Quantity	Pr M
0	McKeesport	NaN	Jessica Myrick	Small Business	0.10	01/01/2012	28774	High	32	
1	Bowie	NaN	Matt Collister	Home Office	0.08	01/01/2012	13729	Not Specified	9	
2	Napa	NaN	Alan Schoenberger	Corporate	0.00	02/01/2012	37537	Low	4	

	Customer Name	Customer Segment	Discount
0	Jessica Myrick	Small Business	0.10

	Customer Name	Customer Segment	Discount
1	Matt Collister	Home Office	0.08
2	Alan Schoenberger	Corporate	0.00
3	Elizabeth Moffitt	Consumer	0.08
4	Alan Schoenberger	Corporate	0.07

📝 Hinglish: Pehli 5 rows aur sirf 3 specific columns nikal liye by position.

💰 5. Real-Life Example: Employee Data

In [75]:

```
data = {
    "Employee": ["Raj", "Simran", "Aman", "Neha"],
    "Salary": [50000, 60000, 70000, 80000],
    "Department": ["HR", "IT", "Finance", "HR"]
}

df = pd.DataFrame(data)

# Single row by position
print(df.iloc[2])

# Multiple rows + specific cols
print("\n", df.iloc[1:3, [0, 1]])
```

```
Employee         Aman
Salary          70000
Department    Finance
Name: 2, dtype: object

   Employee  Salary
1    Simran   60000
2      Aman   70000
```

📝 Hinglish: Position 2 wali row alag nikal li. Aur 1:3 rows ke Employee + Salary columns ek saath le liye.

📌 Difference between .loc vs .iloc

📖 English:

- **.loc** → Select rows/columns by **labels (names)**.
- **.iloc** → Select rows/columns by **index position (numbers)**.

🗣️ Hinglish:

- **.loc** → Naam se data nikalta hai (label-based).
- **.iloc** → Number se data nikalta hai (position-based).

Main Point of Difference:

👉 **.loc** works with labels, while **.iloc** works with integer positions.

📌 (iv) What is Conditional Selection?

📏 Definition — Filter Rows Based on Condition(s)

📖 English:

Conditional selection in **Pandas** means filtering rows based on certain conditions on column values. It allows you to work only with the rows that satisfy your condition(s).

- **Single condition** → Returns a **DataFrame** with rows matching the condition.
- **Multiple conditions** → Combine using **& (and)** / **| (or)**.

🗣️ Hinglish:

Conditional selection matlab dataset me se sirf wahi rows nikalna jo condition pass kare.

👉 Ek condition: sirf **age > 25** wale rows.

👉 Multiple condition: **age > 25** aur **city == "Delhi"**.

🔹 Syntax:

```
Single condition
df[df["col"] > value]


Multiple conditions
df[(df["col1"] > value1) & (df["col2"] == value2)]


Using OR condition
df[(df["col1"] > value1) | (df["col2"] == value2)]
```

- **Parameters**:

  👉 **col / col1, col2** → column names

  👉 **value / value1, value2** → comparison value

- **Returns** → DataFrame (rows satisfying the condition)

📌 Real-Life Examples of Conditional Selection

🍎 1. Single Condition

In [77]:

```
import pandas as pd

data = {"Name": ["Amit", "Ravi", "Simran"],
        "Age": [23, 27, 30]}

df = pd.DataFrame(data)
print(df[df['Age'] > 25])
```

```
     Name  Age
1    Ravi   27
2  Simran   30
```

📝 Hinglish: Sirf Age > 25 wale rows nikal liye.

🚗 2. Multiple Conditions (AND)

In [80]:

```
data = {"Name": ["Amit", "Ravi", "Simran"],
        "Age": [23, 27, 30],
        "City": ["Delhi", "Delhi", "Mumbai"]}

df = pd.DataFrame(data)
print(df[(df["Age"] > 25) & (df["City"]=="Delhi")])
```

```
   Name  Age   City
1  Ravi   27  Delhi
```

📝 Hinglish: Sirf Age > 25 **aur** City Delhi wale rows select hue.

📊 3. Multiple Conditions (OR)

In [81]:

```
print(df[(df["Age"] > 28) | (df["City"] == "Delhi")])
```

```
     Name  Age    City
0    Amit   23   Delhi
1    Ravi   27   Delhi
2  Simran   30  Mumbai
```

📝 Hinglish: Rows jisme Age > 28 **ya** City Delhi ho, woh nikal liye.

⚽ 4. Conditional + Column Selection

In [82]:

```
print(df.loc[df["Age"] > 25, ["Name", "City"]])
```

```
     Name    City
1    Ravi   Delhi
2  Simran  Mumbai
```

📝 Hinglish: Condition apply karke sirf Name aur City columns nikal liye.

💰 5. Real-Life Example: Employee Data

In [87]:

```
data = {
    "Employee": ["Raj", "Simran", "Aman", "Neha"],
    "Salary": [50000, 60000, 70000, 80000],
    "Department": ["HR", "IT", "Finance", "HR"]
}

df = pd.DataFrame(data)

# Salary > 60000
print(df[df["Salary"] > 60000])

# Salary > 60000 AND Department == "HR"
print("\n",df[(df["Salary"] > 60000) & (df["Department"] == "HR")])
```

```
  Employee  Salary Department
2     Aman   70000    Finance
3     Neha   80000         HR

   Employee  Salary Department
3     Neha   80000         HR
```

📝 Hinglish: Condition ke basis pe Employee rows filter ho gaye.

📌 (v) What is Slicing?

📏 Definition — Slice Rows and Columns like Lists/Arrays

📖 English:

Slicing in **Pandas** allows you to extract a subset of rows or columns just like Python lists or arrays. It’s useful when you want a specific range of data instead of the whole dataset.

- **Row slicing** → Selects rows based on position or index range.
- **Column slicing** → Selects columns based on integer positions using **.iloc**.

🗣️ Hinglish:

Slicing matlab dataset me se ek range ya subset nikalna, bilkul Python list jaisa.

👉 **df[0:5]** → pehli 5 rows

👉 **df.iloc[:, 0:2]** → pehle 2 columns

🔹 Syntax:

```
Slice rows by index range
df[start:end]


Slice rows + columns by position
df.iloc[row_start:row_end, col_start:col_end]
```

- **Parameters**:

👉 **start:end** → row numbers (0-based, end exclusive)

👉 **row_start:row_end** → rows

👉 **col_start:col_end** → columns

- **Returns** → DataFrame (subset of rows/columns)

📌 Real-Life Examples of Slicing

🍎 1. Slice Rows Only

In [4]:

```python
import pandas as pd

data = {"Name": ["Amit", "Ravi", "Simran", "Neha", "Raj"],
        "Age": [23, 27, 30, 25, 28]}

df = pd.DataFrame(data)
print(df[0:3])
```

```
     Name  Age
0    Amit   23
1    Ravi   27
2  Simran   30
```

📝 Hinglish: Index 0 se 2 tak ke rows nikal liye (3 exclusive).

🚗 2. Slice Rows + Columns using iloc

In [8]:

```python
print(df.iloc[0:3, 0:1])
```

```
     Name
0    Amit
1    Ravi
2  Simran
```

📝 Hinglish: First 3 rows aur first column select kiya.

📊 3. Slice Last Rows

In [11]:

```
print(df[-2:])
```

```
   Name  Age
3  Neha   25
4   Raj   28
```

📝 Hinglish: Last 2 rows nikal liye, Python-style negative indexing.

⚽ 4. Slice Specific Columns

In [16]:

```
print(df.iloc[:, -1:])
```

```
   Age
0   23
1   27
2   30
3   25
4   28
```

📝 Hinglish: Saare rows ke liye sirf second column (Age) select kiya.

💰 5. Real-Life Example: Employee Data

In [22]:

```
data = {
    "Employee": ["Raj", "Simran", "Aman", "Neha"],
    "Salary": [50000, 60000, 70000, 80000],
    "Department": ["HR", "IT", "Finance", "HR"]
}

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

# First 2 rows
print("\n",df[:2])

# First 2 rows + first 2 columns
print("\n", df.iloc[0:2, 0:2])

# All rows, only salary column
print("\n",df.iloc[:, 1:2])
```

```
  Employee  Salary Department
0      Raj   50000         HR
1   Simran   60000         IT
2     Aman   70000    Finance
3     Neha   80000         HR

   Employee  Salary Department
0      Raj   50000         HR
1   Simran   60000         IT

   Employee  Salary
0      Raj   50000
1   Simran   60000
```

```
   Salary
0   50000
1   60000
2   70000
3   80000
```

📝 Hinglish: Different slicing techniques use karke rows aur columns efficiently extract kiye.

📌 (vi) What is Set Index?

📏 Definition — Set a Column as Index

📖 English:

Setting an index in **Pandas** means replacing the default integer index (0,1,2...) with one of the columns in your DataFrame. It's useful for faster lookups, aligning data, or when a column uniquely identifies each row.

🗣️ Hinglish:

Set index matlab ek column ko primary label bana dena row ke liye.
👉 **df.set_index("id")** → ab "id" column row ka main label ban gaya.
👉 Access rows easily by that index.

- 🔹 Syntax:

```
Set a column as index
df.set_index("col_name", inplace=False)


inplace=True → changes original DataFrame
```

- **Parameters**:
  👉 **col_name** → column name jo index banega
  👉 **inplace** → True: original DF change, False: return new DF

- **Returns** → DataFrame with new index (unless inplace=True)

📌 Real-Life Examples of Set Index

In [23]:

```python
import pandas as pd

data = {"id": [101, 102, 103], "Name": ["Amit", "Ravi", "Simran"], "Age": [23, 27, 30]}
df = pd.DataFrame(data)

# Set 'id' as index
df2 = df.set_index("id")
print(df2)
```

```
       Name  Age
id
101    Amit   23
102    Ravi   27
103  Simran   30
```

📝 Hinglish: Column "id" ab row ka index ban gaya.

🚗 2. Using inplace=True

In [24]:

```python
df.set_index("id", inplace=True)
print(df)
```

```
       Name  Age
id
101    Amit   23
102    Ravi   27
103  Simran   30
```

📝 Hinglish: Original DataFrame ab "id" ke saath updated ho gaya.

📊 3. Access Rows by New Index

In [25]:

```python
print(df.loc[102])
```

```
Name    Ravi
Age       27
Name: 102, dtype: object
```

📝 Hinglish: Index 102 wali row easily fetch ki.

⚽ 4. Reset Index

In [26]:

```python
df.reset_index(inplace=True)
df
```

Out[26]:

	id	Name	Age
0	101	Amit	23
1	102	Ravi	27
2	103	Simran	30

📝 Hinglish: Index wapas default integer me convert ho gaya.

💰 5. Real-Life Example: Employee Data

In [28]:

```
data = {
    "EmpID": [1, 2, 3, 4],
    "Employee": ["Raj", "Simran", "Aman", "Neha"],
    "Salary": [50000, 60000, 70000, 80000]
}

df = pd.DataFrame(data)

# Set EmpID as index
df.set_index("EmpID", inplace=True)
print(df)

# Access Employee with EmpID = 3
print("\n", df.loc[3])
```

```
      Employee  Salary
EmpID
1          Raj   50000
2       Simran   60000
3         Aman   70000
4         Neha   80000

 Employee     Aman
Salary      70000
Name: 3, dtype: object
```

📝 Hinglish: EmpID ko index bana ke specific employee easily access kiya.

📌 (vii) What is Reset Index?

📏 Definition — Reset Index Back to Default 0..n

📖 English:

Resetting an index in **Pandas** means reverting the DataFrame's index back to the default integer range (0,1,2...). It's useful when you no longer want a custom index or after dropping rows that mess up the sequence.

🗣️ Hinglish:

Reset index matlab DataFrame ka index wapas normal 0,1,2... sequence me la dena.
👉 **df.reset_index()** → custom index hat gaya, default integer index wapas aa gaya.

🔹 Syntax:

```
Reset index
df.reset_index(inplace=False)


inplace=True → changes original DataFrame
```

- **Parameters**:

👉 **inplace** → True: original DF change, False: return new DF

👉 **drop** → True: remove old index completely, False: old index becomes column

- **Returns** → DataFrame with default integer index (unless inplace=True)

📌 Real-Life Examples of Reset Index

🍎 1. Basic Example

In [31]:

```python
import pandas as pd

data = {"id": [101, 102, 103], "Name": ["Amit", "Ravi", "Simran"], "Age": [23, 27, 30]}
df = pd.DataFrame(data)

# Set 'id' as index
df.set_index("id", inplace=True)
print(df)

# Reset Index
df_reset = df.reset_index()
print("\n", df_reset)
```

```
       Name  Age
id
101    Amit   23
102    Ravi   27
103  Simran   30

     id    Name  Age
0  101    Amit   23
1  102    Ravi   27
2  103  Simran   30
```

📝 Hinglish: Index "id" hata ke default 0..n wapas aa gaya.

🚗 2. Using inplace=True

In [32]:

```
df.reset_index(inplace=True)
print(df)
```

```
    id    Name  Age
0  101    Amit   23
1  102    Ravi   27
2  103  Simran   30
```

📝 Hinglish: Original DataFrame ab default integer index me updated ho gaya.

📊 3. Drop Old Index Column

In [33]:

```
df.set_index("id", inplace=True)
print(df)

df.reset_index(inplace=True)
print("\n", df)
```

```
       Name  Age
id
101    Amit   23
102    Ravi   27
103  Simran   30

      id    Name  Age
0  101    Amit   23
1  102    Ravi   27
2  103  Simran   30
```

📝 Hinglish: Old index column completely remove ho gaya, sirf default integer index reh gaya.

⚽ 4. Real-Life Example: Employee Data

In [35]:

```
data = {
    "EmpID": [1, 2, 3, 4],
    "Employee": ["Raj", "Simran", "Aman", "Neha"],
    "Salary": [50000, 60000, 70000, 80000]
}

df = pd.DataFrame(data)

# Set EmpID as index
df.set_index("EmpID", inplace=True)
print(df)

# Reset index back
df.reset_index(inplace=True)
print("\n", df)
```

```
      Employee  Salary
EmpID
1          Raj   50000
2       Simran   60000
3         Aman   70000
4         Neha   80000
```

```
   EmpID Employee  Salary
0      1      Raj   50000
1      2   Simran   60000
2      3     Aman   70000
3      4     Neha   80000
```

📝 Hinglish: EmpID ka custom index hata ke default 0..n wapas aa gaya.

5. 🧹 Data Cleaning – Prepare Your Dataset

Operation	Definition	Syntax / Example

Handle Missing Values	Remove or fill missing data.	`df.dropna()`
		`df.fillna(0)`
		`df.fillna(method="ffill'`
		`df.fillna(method="bfill'`

Remove Duplicates	Drop duplicate rows from DataFrame.	`df.drop_duplicates()`
Convert Data Types	Change column data type.	`df["col"] = df["col"].astype(float)`
Rename Columns	Rename column(s) or index.	`df.rename(columns={"old":"new"})`

String Operations	Perform operations on string columns.	`df["name"].str.upper()`
		`df["email"].str.contains`

Apply Functions	Apply custom or lambda functions.	`df["col"].apply(lambda x: x*2)`
		`df.apply(np.sum, axis=0)`

📌 (i) What is Handle Missing Values?

📏 Definition — Remove or Fill Missing Data

📖 English:

Handling missing values in **Pandas** means dealing with **NaN (Not a Number)** entries in your dataset. You can either remove them (drop rows/columns) or fill them with specific values/methods. This is important because missing values can break calculations or give wrong results.

🗣️ Hinglish:

Handle missing values matlab dataset me jo **NaN/empty** values hai unko manage karna.
👉 **Drop kar do** → row/column hatao
👉 **Fill kar do** → 0 ya last/next value se
👉 Isse analysis sahi aur consistent hota hai.

🔹 Syntax:

```
Drop rows with NaN
df.dropna()         # By default axis=0 hota hai → row drop hoti hai jisme NaN hai.
df.dropna(axis=0)  # Rows drop karna (axis=0)
df.dropna(axis=1)  # Columns drop karna (axis=1)


Rows drop → df.dropna() ya df.dropna(axis=0)


Columns drop → df.dropna(axis=1)


Fill NaN with specific value
df.fillna(0)


Forward fill (copy previous value)
df.fillna(method="ffill", axis=0)   # ↓ column ke andar upar se niche fill
df.fillna(method="ffill", axis=1)   # → row ke andar left se right fill


Backward fill (copy next value)
df.fillna(method="bfill", axis=0)   # ↑ column ke andar niche se upar fill
df.fillna(method="bfill", axis=1)   # ← row ke andar right
se left fill
```

- **Parameters**:

  👉 **value** → replace missing with given value

  👉 **method** → ffill (forward fill), bfill (backward fill)

  👉 **axis** → **0**: fill vertically (columns), **1**: fill horizontally (rows)

  👉 **inplace** → True: update original, False: return new

- **Returns** → DataFrame with missing values handled

📌 Real-Life Examples of Handle Missing Values

🍎 1. Drop Rows with NaN (axis=0, default)

In [5]:

```python
import pandas as pd

data = {"Name": ["Amit", "Ravi", None],
        "Age": [23, None, 30]}

df = pd.DataFrame(data)

print("Original:\n",df)
print("\nDrop NaN Rows:\n", df.dropna(axis=0))
```

```
Original:
    Name   Age
0  Amit  23.0
1  Ravi   NaN
2  None  30.0

Drop NaN Rows:
    Name   Age
0  Amit  23.0
```

📝 Hinglish: Jaha bhi NaN mila, woh row delete ho gayi.

🚗 2. Drop Columns with NaN (axis=1)

In [7]:

```python
print("\n Drop NaN Columns:\n", df.dropna(axis=1))
```

```
 Drop NaN Columns:
 Empty DataFrame
Columns: []
Index: [0, 1, 2]
```

📝 Hinglish: Jaha bhi NaN mila, woh column delete ho gaya.

📊 3. Fill NaN with a Specific Value

In [8]:

```
df_filled =df.fillna(0)
print("\nFill NaN with 0:\n", df_filled)
```

```
Fill NaN with 0:
    Name   Age
0  Amit  23.0
1  Ravi   0.0
2     0  30.0
```

📝 Hinglish: Sabhi NaN ko 0 se replace kar diya.

⚽ 4. Forward Fill (ffill → axis=0 default)

In [13]:

```
df_ffill = df.fillna(method="ffill", axis=0)
print("\nForward Fill (row-wise):\n", df_ffill)

df_ffill_col = df.fillna(method="ffill", axis=1)
print("\nForward Fill (column-wise):\n", df_ffill_col)
```

```
Forward Fill (row-wise):
    Name   Age
0  Amit  23.0
1  Ravi  23.0
2  Ravi  30.0

Forward Fill (column-wise):
    Name   Age
0  Amit  23.0
1  Ravi  Ravi
2   NaN  30.0
C:\Users\gkuir\AppData\Local\Temp\ipykernel_21440\173582138.py:1: FutureWarning: DataFra
me.fillna with 'method' is deprecated and will raise in a future version. Use obj.ffill
() or obj.bfill() instead.
  df_ffill = df.fillna(method="ffill", axis=0)
C:\Users\gkuir\AppData\Local\Temp\ipykernel_21440\173582138.py:4: FutureWarning: DataFra
me.fillna with 'method' is deprecated and will raise in a future version. Use obj.ffill
() or obj.bfill() instead.
  df_ffill_col = df.fillna(method="ffill", axis=1)
C:\Users\gkuir\AppData\Local\Temp\ipykernel_21440\173582138.py:4: FutureWarning: Downcas
ting object dtype arrays on .fillna, .ffill, .bfill is deprecated and will change in a f
uture version. Call result.infer_objects(copy=False) instead. To opt-in to the future be
havior, set `pd.set_option('future.no_silent_downcasting', True)`
  df_ffill_col = df.fillna(method="ffill", axis=1)
```

📝 Hinglish:

- axis=0 → NaN ko upar wali value se bhar diya (column ke andar).
- axis=1 → NaN ko left wali value se bhar diya (row ke andar).

🎯 5. Backward Fill (bfill)

In [14]:

```
df_bfill = df.fillna(method="bfill", axis=0)
print("\nBackward Fill (row-wise):\n", df_bfill)

df_bfill_col = df.fillna(method="bfill", axis=1)
print("\nBackward Fill (column-wise):\n", df_bfill_col)
```

```
Backward Fill (row-wise):
    Name   Age
0  Amit  23.0
1  Ravi  30.0
2  None  30.0

Backward Fill (column-wise):
    Name   Age
0  Amit  23.0
1  Ravi   NaN
2  30.0  30.0
C:\Users\gkuir\AppData\Local\Temp\ipykernel_21440\492467896.py:1: FutureWarning: DataFra
me.fillna with 'method' is deprecated and will raise in a future version. Use obj.ffill
() or obj.bfill() instead.
  df_bfill = df.fillna(method="bfill", axis=0)
C:\Users\gkuir\AppData\Local\Temp\ipykernel_21440\492467896.py:4: FutureWarning: DataFra
me.fillna with 'method' is deprecated and will raise in a future version. Use obj.ffill
() or obj.bfill() instead.
  df_bfill_col = df.fillna(method="bfill", axis=1)
C:\Users\gkuir\AppData\Local\Temp\ipykernel_21440\492467896.py:4: FutureWarning: Downcas
ting object dtype arrays on .fillna, .ffill, .bfill is deprecated and will change in a f
uture version. Call result.infer_objects(copy=False) instead. To opt-in to the future be
havior, set `pd.set_option('future.no_silent_downcasting', True)`
  df_bfill_col = df.fillna(method="bfill", axis=1)
```

📝 Hinglish:

- axis=0 → NaN ko neeche wali value se bhar diya (column ke andar).
- axis=1 → NaN ko right wali value se bhar diya (row ke andar).

💰 6. Real-Life Example: Employee Data

In [15]:

```
data = {
    "Employee": ["Raj", "Simran", "Aman", "Neha"],
    "Salary": [50000, None, 70000, None],
    "Department": ["HR", "IT", None, "Finance"]
}

df = pd.DataFrame(data)
print("Original:\n", df)

# Fill Salary NaN with 0
print("\nFill Salary with 0:\n", df.fillna({"Salary": 0}))

# Drop columns with NaN values
print("\nDrop NaN Columns:\n", df.dropna(axis=1))
```

```
# Forward fill Department
print("\nForward Fill Department:\n", df.fillna(method="ffill"))
```

```
Original:
   Employee   Salary Department
0       Raj  50000.0         HR
1    Simran      NaN         IT
2      Aman  70000.0       None
3      Neha      NaN    Finance

Fill Salary with 0:
   Employee   Salary Department
0       Raj  50000.0         HR
1    Simran      0.0         IT
2      Aman  70000.0       None
3      Neha      0.0    Finance

Drop NaN Columns:
   Employee
0       Raj
1    Simran
2      Aman
3      Neha

Forward Fill Department:
   Employee   Salary Department
0       Raj  50000.0         HR
1    Simran  50000.0         IT
2      Aman  70000.0         IT
3      Neha  70000.0    Finance
C:\Users\gkuir\AppData\Local\Temp\ipykernel_21440\2722815281.py:17: FutureWarning: DataF
rame.fillna with 'method' is deprecated and will raise in a future version. Use obj.ffil
l() or obj.bfill() instead.
  print("\nForward Fill Department:\n", df.fillna(method="ffill"))
```

📝 Hinglish: Salary me NaN ko 0 kiya, NaN wala column drop kiya, aur Department ko forward-fill se bhar diya.

📌 (ii) What is Remove Duplicates?

📏 Definition — Drop Duplicate Rows from DataFrame

📖 English:

Removing duplicates in **Pandas** means dropping repeated rows from your DataFrame, so that only **unique rows** remain. It helps in cleaning data, avoiding redundancy, and ensuring correct analysis.

🗣️ Hinglish:

Remove duplicates matlab DataFrame me jo **rows bar-bar repeat** ho rahi hain, unko hata dena.
👉 **df.drop_duplicates()** → sirf unique rows bachi rahengi.

👉 Useful jab dataset me duplicate entries galti se aa gayi ho.

🔹 Syntax:

```
Drop duplicate rows
df.drop_duplicates(subset=None, keep='first', inplace=False)
```

- **Parameters**:

👉 **subset** → specify column(s) to check duplicates (default: all columns)

👉 **keep** → 'first' (keep first occurrence), 'last' (keep last occurrence), False (drop all duplicates)

👉 **inplace** → True: update original DataFrame, False: return new one

- **Returns** → DataFrame with duplicates removed

📌 Real-Life Examples of Remove Duplicates

🍎 1. Basic Example

In [16]:

```python
import pandas as pd

data = {
    "id": [101, 102, 103, 101, 102],
    "Name": ["Amit", "Ravi", "Simran", "Amit", "Ravi"],
    "Age": [23, 27, 30, 23, 27]
}

df = pd.DataFrame(data)
print("Original:\n", df)

# Remove duplicates
df_unique = df.drop_duplicates()
print("\nAfter Removing Duplicates:\n", df_unique)
```

```
Original:
      id    Name  Age
0  101    Amit   23
1  102    Ravi   27
2  103  Simran   30
3  101    Amit   23
4  102    Ravi   27

After Removing Duplicates:
      id    Name  Age
0  101    Amit   23
```

```
1  102    Ravi   27
2  103  Simran   30
```

📝 Hinglish: Duplicate rows gayab ho gayi, sirf unique rows bachi.

🚗 2. Keep Last Duplicate

In [17]:

```
df_last = df.drop_duplicates(keep="last")
print("\nKeep Last Duplicate:\n", df_last)
```

```
Keep Last Duplicate:
     id    Name  Age
2  103  Simran   30
3  101    Amit   23
4  102    Ravi   27
```

📝 Hinglish: Same rows me se last wali rakhi, baaki delete ho gayi.

📊 3. Drop All Duplicates (No keep)

In [19]:

```
df_none = df.drop_duplicates(keep=False)
print("\nDrop All Duplicates:\n", df_none)
```

```
Drop All Duplicates:
     id    Name  Age
2  103  Simran   30
```

📝 Hinglish: Jitni bhi duplicate thi unko pura hata diya, sirf unique rows bachi.

⚽ 4. Remove Duplicates Based on Specific Column

In [20]:

```
df_name = df.drop_duplicates(subset=["Name"])
print("\nDrop Duplicates by Name:\n", df_name)
```

```
Drop Duplicates by Name:
     id    Name  Age
0  101    Amit   23
1  102    Ravi   27
2  103  Simran   30
```

📝 Hinglish: Sirf `Name` column ke basis par duplicate rows hata di gayi.

💰 5. Real-Life Example: Customer Orders

In [22]:

```
data = {
    "OrderID": [1, 2, 3, 3, 4, 5, 5],
    "Customer": ["Raj", "Simran", "Aman", "Aman", "Neha", "Raj", "Raj"],
    "Amount": [250, 300, 150, 150, 500, 250, 250]
}

df = pd.DataFrame(data)
print("Original Orders:\n", df)

# Remove duplicate orders
```

```
print("\nUnique Orders:\n", df.drop_duplicates())

# Remove duplicates by Customer, keep last
print("\nUnique Customers (last kept):\n", df.drop_duplicates(subset=["Customer"], keep=
```

```
Original Orders:
    OrderID Customer  Amount
0         1      Raj     250
1         2   Simran     300
2         3     Aman     150
3         3     Aman     150
4         4     Neha     500
5         5      Raj     250
6         5      Raj     250

Unique Orders:
    OrderID Customer  Amount
0         1      Raj     250
1         2   Simran     300
2         3     Aman     150
4         4     Neha     500
5         5      Raj     250

Unique Customers (last kept):
    OrderID Customer  Amount
1         2   Simran     300
3         3     Aman     150
4         4     Neha     500
6         5      Raj     250
```

📝 Hinglish: Orders me duplicate rows thi, unko hata kar clean dataset ban gaya.

📌 (iii) What is Convert Data Types?

📏 Definition — Change Column Data Type

📖 English:

Converting data types in Pandas means changing the type of a column (like **integer → float**, **string → datetime**, etc.).

It's useful when you want consistency in data, accurate calculations, and proper analysis.

🗣️ Hinglish:

Convert data types matlab DataFrame ke **column ka type badalna**.

👉 int ko float me, ya string ko datetime me convert karna.

👉 **df["col"] = df["col"].astype(float)** → column ka datatype float ho gaya.

👉 Useful jab galti se column wrong type me load ho jata hai.

🔹 Syntax:

```
Convert column to specific datatype
df["col"] = df["col"].astype(datatype)
```

- **Parameters**:

  👉 **datatype** → specify data type (int, float, str, bool, 'category', 'datetime64[ns]', etc.)

  👉 **errors**:

  ✔️ **'raise'** → error throw karega agar convert na ho paya (default)

  ✔️ **'ignore'** → conversion fail hua to original value reh jaayegi

  ✔️ **'coerce'** → conversion fail hua to NaT/NaN assign ho jayega

- **Returns** → Series ya DataFrame with converted datatype

📌 Real-Life Examples of Convert Data Types

🍎 1. Basic Example: int → float

In [26]:

```python
import pandas as pd

data = {"id": [101, 102, 103], "Age": [23, 27, 30]}
df = pd.DataFrame(data)
print("Original:\n", df.dtypes)

# Conver Age to float
df["Age"] = df["Age"].astype(float)
print("\nAfter Conversion:\n", df.dtypes)
```

```
Original:
 id     int64
Age    int64
dtype: object

After Conversion:
 id       int64
Age    float64
dtype: object
```

📝 Hinglish: Age column int se float ban gaya.

🚗 2. String → Integer

In [29]:

```python
df = pd.DataFrame({"Marks": ["85", "90", "95"]})
print("Original:\n", df.dtypes)

# Convert to int
```

```
df["Marks"] = df["Marks"].astype(int)
print("\nAfter Conversion: \n", df.dtypes)
```

```
Original:
 Marks    object
dtype: object

After Conversion:
 Marks    int64
dtype: object
```

📝 Hinglish: String marks ko integer me convert kar diya.

📊 3. String → Datetime

In [33]:

```
df = pd.DataFrame({"Date": ["2023-01-01", "2023-02-15", "2023-03-20"]})
print("Original:\n", df.dtypes)
print(df)

# Convert to datetime
df['Date'] = pd.to_datetime(df['Date'])
print("\nAfter Conversion:\n", df.dtypes)
```

```
Original:
 Date    object
dtype: object
         Date
0  2023-01-01
1  2023-02-15
2  2023-03-20

After Conversion:
 Date    datetime64[ns]
dtype: object
```

📝 Hinglish: Date column ab datetime type ban gaya.📝 Hinglish: Date column ab datetime type ban gaya.

⚽ 4. Multiple Columns Conversion

In [36]:

```
data = {"Roll": ["1", "2", "3"], "Marks": ["80.5", "90.2", "75.3"]}
df = pd.DataFrame(data)
print(df.dtypes)

# Convert Roll → int, Marks → float
df = df.astype({"Roll":int, "Marks":float})
print("\n", df.dtypes)
```

```
Roll     object
Marks    object
dtype: object

 Roll       int64
Marks    float64
dtype: object
```

📝 Hinglish: Ek hi line me multiple columns ka datatype change kar diya.

💰 5. Real-Life Example: Sales Data

In [37]:

```
sales = {
    "OrderID": ["1", "2", "3"],
    "Amount": ["250.5", "300.0", "150.75"],
    "Date": ["2023-05-01", "2023-05-02", "2023-05-03"]
}

df = pd.DataFrame(sales)
print("Before:\n", df.dtypes)

# Convert Amount → float, Date → datetime
df = df.astype({"Amount": float})
df["Date"] = pd.to_datetime(df["Date"])

print("\nAfter:\n", df.dtypes)
```

```
Before:
 OrderID    object
Amount     object
Date       object
dtype: object

After:
 OrderID            object
Amount            float64
Date       datetime64[ns]
dtype: object
```

📝 Hinglish: Sales dataset me Amount float aur Date datetime me convert kar diya.

📌 (iv) What is Rename Columns?

📏 Definition — Rename Column(s) or Index

📖 English:

Renaming columns in **Pandas** means changing the name of one or more columns (or index labels) in a DataFrame.

It's useful for making column names more **meaningful, standardized, or analysis-friendly**.

🗣️ Hinglish:

Rename columns matlab DataFrame ke **column ka naam badalna**.

👉 **df.rename(columns={"old":"new"})** → column ka naam change ho gaya.

👉 Useful jab dataset me confusing ya messy column names ho.

🔹 Syntax:

```
Rename columns
df.rename(columns={"old_name": "new_name"}, inplace=False)


Rename index
df.rename(index={old_index: new_index}, inplace=False)
```

- **Parameters**:

  👉 **columns** → dict: specify which columns to rename

  👉 **index** → dict: specify which index labels to rename

  👉 **inplace** → True: modify original DataFrame, False: return new one

- **Returns** → DataFrame with updated column/index names

📌 Real-Life Examples of Rename Columns

🍎 1. Basic Example: Rename One Column

In [38]:

```
import pandas as pd

data = {"id": [101, 102, 103], "Name": ["Amit", "Ravi", "Simran"]}
df = pd.DataFrame(data)

# Rename Column
df_renamed = df.rename(columns={"id":"ID"})
print(df_renamed)
```

```
    ID    Name
0  101    Amit
1  102    Ravi
2  103  Simran
```

📝 Hinglish: "id" column ka naam "ID" ban gaya.

🚗 2. Rename Multiple Columns

In [39]:

```
df_multi = df.rename(columns= {"id":"ID", "Name":"Full_Name"})
print(df_multi)
```

```
    ID Full_Name
0  101      Amit
1  102      Ravi
2  103    Simran
```

📝 Hinglish: Ek saath multiple columns ke naam change ho gaye.

📊 3. Rename Index Labels

In [40]:

```
df_index = df.rename(index={0:"First", 1:"Second"})
print(df_index)
```

```
         id    Name
First   101    Amit
Second  102    Ravi
2       103  Simran
```

⚽ 4. Using inplace=True

In [42]:

```
df.rename(columns={"Name":"Employees"}, inplace=True)
print(df)
```

```
    id Employees
0  101      Amit
1  102      Ravi
2  103    Simran
```

📝 Hinglish: Original DataFrame me hi column ka naam change ho gaya.

💰 5. Real-Life Example: Sales Data Cleanup

In [43]:

```
sales = {
    "OrderID": [1, 2, 3],
    "CustName": ["Raj", "Simran", "Aman"],
    "Amt": [250, 300, 150]
}

df = pd.DataFrame(sales)
print("Before:\n", df)

# Rename messy column names
df = df.rename(columns={"CustName": "Customer", "Amt": "Amount"})
print("\nAfter:\n", df)
```

```
Before:
    OrderID CustName  Amt
0        1      Raj  250
1        2   Simran  300
2        3     Aman  150

After:
    OrderID Customer  Amount
0        1      Raj     250
1        2   Simran     300
2        3     Aman     150
```

📝 Hinglish: Sales dataset me short/messy names ko proper names se replace kar diya.

📌 (v) What are String Operations?

📏 Definition — Perform String Operations on DataFrame Columns

📖 English:

String operations in **Pandas** allow you to manipulate text data stored in columns (like names, emails, categories).
You can make everything **uppercase/lowercase**, check for substrings, split text, replace characters, and much more.

🗣️ Hinglish:

String operations matlab DataFrame ke **text wale columns** pe direct functions lagana.
👉 **df["name"].str.upper()** → sab naam uppercase ho gaye.
👉 Useful jab **data cleaning** karna ho ya text ko analysis-friendly banana ho.

🔹 Syntax:

```
Uppercase
df["col"].str.upper()


Lowercase
df["col"].str.lower()


Strip (remove extra spaces)
df["col"].str.strip()


Contains (check substring)
df["col"].str.contains("abc")


Replace
df["col"].str.replace("old", "new")


Split
df["col"].str.split(" ")
```

- **Parameters**:

  👉 **pat** → pattern ya substring (for contains/replace/split)

  👉 **case** → True/False, case-sensitive search

  👉 **regex** → whether to treat pattern as regex (default True)

- **Returns** → Series with transformed string values

📌 Real-Life Examples of String Operations

🍎 1. Uppercase & Lowercase

In [4]:

```
import pandas as pd

data = {"Name": ["Amit", "ravi", "Simran"], "Email": ["amit@gmail.com", "ravi@yahoo.com"
df = pd.DataFrame(data)

# Uppercase
print(df["Name"].str.upper())

# Lowercase
print("\n",df["Name"].str.lower())
```

```
0      AMIT
1      RAVI
2    SIMRAN
Name: Name, dtype: object

 0      amit
1      ravi
2    simran
Name: Name, dtype: object
```

📝 Hinglish: Naam sab uppercase ya lowercase me convert ho gaye.

🚗 2. Strip Extra Spaces

In [5]:

```
df2 = pd.DataFrame({"City": [" Delhi ", " Mumbai  ", "Kolkata"]})
print(df2["City"].str.strip())
```

```
0      Delhi
1     Mumbai
2    Kolkata
Name: City, dtype: object
```

📝 Hinglish: Extra spaces hat gaye, clean text mil gaya.

📊 3. Check if Email is Gmail

In [6]:

```
print(df["Email"].str.contains("@gmail"))
```

```
0     True
1    False
2     True
Name: Email, dtype: bool
```

📝 Hinglish: True/False batata hai ki email Gmail ka hai ya nahi.

⚽ 4. Replace Text

In [7]:

```
print(df["Email"].str.replace("gmail","Outlook"))
```

```
0      amit@Outlook.com
1        ravi@yahoo.com
2    simran@Outlook.com
Name: Email, dtype: object
```

📝 Hinglish: "gmail" ko "outlook" se replace kar diya.

💰 5. Real-Life Example: Extract Domain from Email

In [14]:

```
print(df)  # Before df view
```

```
     Name              Email     Domain
0    Amit     amit@gmail.com  gmail.com
1    ravi     ravi@yahoo.com  yahoo.com
2  Simran   simran@gmail.com  gmail.com
```

In [12]:

```
df["Domain"]=df["Email"].str.split("@")  # After split view
print(df)
```

```
     Name              Email               Domain
0    Amit     amit@gmail.com    [amit, gmail.com]
1    ravi     ravi@yahoo.com    [ravi, yahoo.com]
2  Simran   simran@gmail.com  [simran, gmail.com]
```

In [13]:

```
df["Domain"]=df["Email"].str.split("@").str[1]   # Af
print(df)
```

```
     Name              Email     Domain
0    Amit     amit@gmail.com  gmail.com
1    ravi     ravi@yahoo.com  yahoo.com
2  Simran   simran@gmail.com  gmail.com
```

📝 Hinglish: Email se domain (gmail.com, yahoo.com) nikal liya.

📌 What is Lambda Function?

📏 Definition — Anonymous / One-Liner Functions

📖 English:

A lambda function in Python is a small, **anonymous (nameless)** function defined using the keyword **lambda**.

It can take **multiple inputs** but must contain only **one expression**, which gets evaluated and returned automatically.

👉 Basically, it’s used when you need a **quick, throwaway function** without formally defining it using **def**.

🗣️ Hinglish:

Lambda function ek chhoti aur bina naam wali function hoti hai jo ek hi line me likhi jaati hai.

👉 **lambda x: x*2** → ye har input ko 2 se multiply karega.

👉 Useful jab **temporary function** chahiye ho aur bar-bar **def** likhna bore lage.

🔹 Syntax:

```
General Syntax
lambda arguments: expression
```

- Rules:

👉 Multiple arguments allowed

👉 Sirf ek hi **expression** hota hai (no multiple statements)

👉 **return** likhne ki zarurat nahi — expression ka result auto return hota hai

- Returns → Single evaluated value from the expression

📌 Real-Life Examples of String Operations

🍎 1. Basic Example: Multiply by 2

In [25]:

```
double = lambda x: x * 2
print(double(5))    # Output: 10
```

```
10
```

📝 Hinglish: Input 5 diya, output 10 aaya.

🚗 2. Multiple Arguments

In [26]:

```
add = lambda x, y: x + y
print(add(3, 7))    # Output: 10
```

```
10
```

📝 Hinglish: 3 aur 7 ko add karke 10 de diya.

📊 3. With map() Function

In [27]:

```
nums = [1, 2, 3, 4, 5]
squared = list(map(lambda x: x**2, nums))
print(squared)    # Output: [1, 4, 9, 16, 25]
```

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

📝 Hinglish: Har number ka square nikal diya.

⚽ 4. With `filter()` Function

In [28]:

```
nums = [10, 15, 20, 25, 30]
even = list(filter(lambda x: x % 2 == 0, nums))
print(even)    # Output: [10, 20, 30]
```

```
[10, 20, 30]
```

📝 Hinglish: Sirf even numbers filter kar liye.

💰 5. With `sorted()` Function

In [29]:

```
data = [("Raj", 25), ("Simran", 22), ("Aman", 30)]
sorted_data = sorted(data, key=lambda x: x[1])
print(sorted_data)    # Output: [('Simran', 22), ('Raj', 25), ('Aman', 30)]
```

```
[('Simran', 22), ('Raj', 25), ('Aman', 30)]
```

📝 Hinglish: Tuple list ko age (2nd element) ke basis pe sort kar diya.

📚 6. Real-Life Example: Salary Bonus

In [30]:

```
salary = [50000, 60000, 70000]
bonus = list(map(lambda x: x + x*0.1, salary))
print(bonus)    # Output: [55000.0, 66000.0, 77000.0]
```

```
[55000.0, 66000.0, 77000.0]
```

📝 Hinglish: Salary me 10% bonus add ho gaya.

📌 (vi) What is Apply Functions?

📏 Definition — Apply Custom or Lambda Functions

📖 English:

In Pandas, **.apply()** lets you apply custom functions (or lambda functions) to each element, row, or column of a DataFrame/Series.

It’s super powerful for doing **transformations, aggregations,** and complex operations that aren’t covered by built-in methods.

🗣️ Hinglish:

.apply() ka matlab hai apne **khud ke function ya lambda** ko DataFrame ya Series pe apply karna.

👉 **df["col"].apply(lambda x: x*2)** → column ke har element ko 2 se multiply kar dega.

👉 Useful jab **built-in methods kaam na kare** ya custom logic lagana ho.

🔹 Syntax:

```
Apply function on Series
df["col"].apply(func)


Apply function on DataFrame
df.apply(func, axis=0)   # column-wise
df.apply(func, axis=1)   # row-wise
```

- **Parameters**:

  👉 **func** → function ya lambda jo apply karna hai

  👉 **axis** → 0: apply function on columns, 1: apply function on rows

- **Returns** → Series ya DataFrame with transformed values

📌 Real-Life Examples of Apply Functions

🍎 1. Apply on Series (Lambda Example)

In [15]:

```
import pandas as pd
import numpy as np

data = {"Numbers": [1, 2, 3, 4, 5]}
df = pd.DataFrame(data)
```

```
# Multiply each element by 2
print(df["Numbers"].apply(lambda x: x*2))
```

```
0     2
1     4
2     6
3     8
4    10
Name: Numbers, dtype: int64
```

📝 Hinglish: Har number ko 2 se multiply kar diya.

🚗 2. Apply on DataFrame (Column-wise Sum)

In [18]:

```
print(df.apply(np.sum, axis = 0))
```

```
Numbers    15
dtype: int64
```

📊 3. Apply Row-wise (Custom Function)

In [19]:

```
data = {"Math": [50, 80, 90], "Science": [70, 60, 100]}
df2 = pd.DataFrame(data)

# Row-wise total
df2["Total"] = df2.apply(lambda row: row["Math"] + row["Science"], axis=1)
print(df2)
```

```
   Math  Science  Total
0    50       70    120
1    80       60    140
2    90      100    190
```

📝 Hinglish: Har row ka Math + Science total nikal liya.

⚽ 4. Using Built-in Function

In [21]:

```
names = pd.DataFrame({"Name": ["amit", "simran", "ravi"]})

# Capitalize first letter of each name
print(names["Name"].apply(str.capitalize))
```

```
0      Amit
1    Simran
2      Ravi
Name: Name, dtype: object
```

📝 Hinglish: Har naam ka first letter capitalize ho gaya.

💰 5. Real-Life Example: Salary Bonus Calculation

In [22]:

```
salary = pd.DataFrame({"Employee": ["Raj", "Simran", "Aman"], "Salary": [50000, 60000, 7

# Add 10% bonus
```

```
salary["With_Bonus"] = salary["Salary"].apply(lambda x: x + x*0.1)
print(salary)
```

```
  Employee  Salary  With_Bonus
0      Raj   50000     55000.0
1   Simran   60000     66000.0
2     Aman   70000     77000.0
```

📝 Hinglish: Salary me 10% bonus add ho gaya.

6. 🔧 Data Manipulation – Modify Your Dataset

Operation	Definition	Syntax / Example

Sort Values	Sort DataFrame by column values.	`df.sort_values("col")`
		`df.sort_values(["col1",' ascending=[True, False])`

Sort Index	Sort DataFrame by index.	`df.sort_index()`

Add / Remove Columns	Insert or drop columns.	`df["new"] = values`
		`df.drop("col", axis=1)`

Insert Column	Insert column at specific position.	`df.insert(1, "new_col", values)`
Replace Values	Replace specific values in DataFrame.	`df.replace({1: "A", 2: "B"})`

Map Values	Map values of Series using dict or function.	`df["col"].map({1:"One", 2:"Two"})`
		`df["col"].map(lambda x: x*2)`

Applymap	Apply function element-wise across entire DataFrame.	`df.applymap(str.upper)`

📌 (i) What is Sort Values?

📏 **Definition — Sort DataFrame by Column Values**

📖 English:

Sorting values in Pandas means arranging rows of a DataFrame based on one or more column values (**ascending** or **descending** order).

👉 **df.sort_values("col")** → sorts by a single column.

👉 **df.sort_values(["col1", "col2"], ascending=[True, False])** → sorts by multiple columns with different orders.

🗣️ Hinglish:

Sort values ka matlab hai DataFrame ke **rows ko column ke values** ke basis pe arrange karna.

👉 **df.sort_values("Age")** → Age ke hisaab se sort ho jaayega.

👉 Useful jab data ko **order me dekhna** ho, jaise **top scorer** ya **lowest price** find karna.

🔹 Syntax:

```
Sort by single column
df.sort_values("col", ascending=True)


Sort by multiple columns
df.sort_values(["col1", "col2"], ascending=[True, False])
```

- **Parameters**:

👉 **by** → column name(s) specify karo jinke basis pe sort karna hai

👉 **ascending** → True (default) ascending, False descending

👉 **inplace** → True → modify original DataFrame, False → return new DataFrame

👉 **na_position** → "last" (default), "first" → NaN pehle aayenge

- **Returns** → Sorted DataFrame

📌 Real-Life Examples of Sort Values

🍎 1. Basic Example: Sort by One Column

In [35]:

```python
import pandas as pd

data = {"Name": ["Amit", "Ravi", "Simran"], "Age": [25, 30, 22]}
df = pd.DataFrame(data)
```

```
#Sort by Age
print(df.sort_values("Age"))
```

```
     Name  Age
2  Simran   22
0    Amit   25
1    Ravi   30
```

📝 Hinglish: Age ke hisaab se ascending order me arrange ho gaya.

🚗 2. Sort in Descending Order

In [36]:

```
print(df.sort_values("Age", ascending=False))
```

```
     Name  Age
1    Ravi   30
0    Amit   25
2  Simran   22
```

📝 Hinglish: Age ke basis pe descending (bade se chhote) order me sort ho gaya.

📊 3. Sort by Multiple Columns

In [39]:

```
data = {"Name": ["Amit", "Ravi", "Simran", "Ravi"], "Age": [25, 30, 22, 30], "Score": [8
df = pd.DataFrame(data)

# Sort by Age ascending, then Score descending
print(df.sort_values(["Age", "Score"], ascending= [True, False]))
```

```
     Name  Age  Score
2  Simran   22     75
0    Amit   25     85
3    Ravi   30     95
1    Ravi   30     90
```

📝 Hinglish: Pehle Age ke basis pe sort hua, phir same Age wale Score ke hisaab se descending order me aaye.

⚽ 4. Sort with NaN Values

In [40]:

```
data = {"Name": ["Amit", "Ravi", "Simran"], "Age": [25, None, 22]}
df = pd.DataFrame(data)

print(df.sort_values("Age", na_position="first"))
```

```
     Name   Age
1    Ravi   NaN
2  Simran  22.0
0    Amit  25.0
```

📝 Hinglish: NaN ko top pe le aaya.

💰 5. Real-Life Example: Sales Data

In [41]:

```
sales = {
    "OrderID": [1, 2, 3, 4],
    "Customer": ["Raj", "Simran", "Aman", "Neha"],
    "Amount": [250, 500, 150, 400]
}

df = pd.DataFrame(sales)

# Sort by Amount (highest sales first)
print(df.sort_values("Amount", ascending=False))
```

```
   OrderID Customer  Amount
1        2   Simran     500
3        4     Neha     400
0        1      Raj     250
2        3     Aman     150
```

📝 Hinglish: Sales data ko Amount ke basis pe highest to lowest arrange kar diya.

📌 (ii) What is Sort Index?

📏 Definition — Sort DataFrame by Index

📖 English:

Sorting index in Pandas means arranging the **rows** or **columns** of a DataFrame based on their **index labels** (**ascending** or **descending** order).

👉 **df.sort_index()** → sorts rows by their index.

👉 **df.sort_index(axis=1)** → sorts columns by their index labels.

🗣️ Hinglish:

Sort index ka matlab hai DataFrame ke **rows ya columns** ko unke **index ke naam/number** ke basis pe arrange karna.

👉 **df.sort_index()** → row indexes ko ascending order me arrange karega.

👉 Useful jab aapko index properly **order me rakhna** ho, especially after **shuffling** ya **concatenation**.

🔹 Syntax:

```
Sort rows by index (default)
df.sort_index(ascending=True)
```

```
Sort columns by index
df.sort_index(axis=1, ascending=True)
```

- **Parameters**:

👉 **axis** → 0 → sort rows, 1 → sort columns

👉 **ascending** → True (default) ascending, False descending

👉 **inplace** → True → modify original DataFrame, False → return new DataFrame

👉 **na_position** → "last" (default), "first"

- **Returns** → Sorted DataFrame

📌 Real-Life Examples of Sort Index

🍎 1. Basic Example: Sort Row Index

In [44]:

```
import pandas as pd

data = {"Name": ["Amit", "Ravi", "Simran"], "Age": [25, 30, 22]}
df = pd.DataFrame(data, index=[2, 0, 1])

print(df.sort_index())
```

```
     Name  Age
0    Ravi   30
1  Simran   22
2    Amit   25
```

📝 Hinglish: Indexes ko ascending order me arrange kar diya.

🚗 2. Sort Row Index in Descending Order

In [45]:

```
print(df.sort_index(ascending=False))
```

```
     Name  Age
2    Amit   25
1  Simran   22
0    Ravi   30
```

📝 Hinglish: Index ko ulte (descending) order me arrange kar diya.

📊 3. Sort Column Index

In [46]:

```
data = {"b": [1, 2, 3], "a": [4, 5, 6], "c": [7, 8, 9]}
df = pd.DataFrame(data)
```

```
print(df.sort_index(axis=1))
```

```
   a  b  c
0  4  1  7
1  5  2  8
2  6  3  9
```

📝 Hinglish: Columns ko unke labels ke ascending order (a → b → c) me arrange kar diya.

⚽ 4. Sort with NaN Index

In [47]:

```
import numpy as np
df_nan = pd.DataFrame({"A": [10, 20], "B": [30, 40]}, index=[np.nan, 2])

print(df_nan.sort_index(na_position="first"))
```

```
      A   B
NaN  10  30
2.0  20  40
```

📝 Hinglish: NaN index ko sabse pehle la diya.

💰 5. Real-Life Example: Sales Data with Random Index

In [48]:

```
sales = {
    "OrderID": [101, 102, 103],
    "Customer": ["Raj", "Simran", "Aman"],
    "Amount": [250, 500, 150]
}

df = pd.DataFrame(sales, index=[5, 2, 8])

# Sort by index
print(df.sort_index())
```

```
   OrderID Customer  Amount
2      102   Simran     500
5      101      Raj     250
8      103     Aman     150
```

📝 Hinglish: Sales data ko index ke basis pe arrange kar diya.

📌 (iii) What is Add / Remove Columns?

📏 Definition — Insert or Drop Columns in DataFrame

📖 English:

In Pandas, you can **add new columns** to a DataFrame by assigning values, and **remove existing columns** using **.drop()**.

👉 **df["new"] = values** → add a new column.

👉 **df.drop("col", axis=1)** → remove a column.

🗣️ Hinglish:

DataFrame me **naye column add karna** ya **purane column hataana** easy hai.

👉 **df["new"] = values** → ek naya column ban gaya.

👉 **df.drop("col", axis=1)** → column remove ho gaya.

👉 Useful jab **data cleaning** aur **feature engineering** kar rahe ho.

🔹 Syntax:

```
Add new column
df["new_col"] = values


Drop single column
df.drop("col", axis=1)


Drop multiple columns
df.drop(["col1", "col2"], axis=1)


Drop inplace (modifies original DataFrame)
df.drop("col", axis=1, inplace=True)
```

- **Parameters**:

  👉 **labels** → Column name(s) jo drop karne hain

  👉 **axis** → 1 (columns), 0 (rows)

  👉 **inplace** → True → modify original DataFrame, False → return new DataFrame

  👉 **errors** → "ignore" → error na de agar column na mile

- **Returns** → New DataFrame (agar inplace=False hai)

📌 Real-Life Examples of Add / Remove Columns

🍎 1. Add New Column with List

```
In [1]:
```

```
import pandas as pd

data = {"Name": ["Amit", "Ravi", "Simran"], "Age": [25, 30, 22]}
df = pd.DataFrame(data)
df["Score"] = [80, 90, 75]
print(df)
```

```
     Name  Age  Score
0    Amit   25     80
1    Ravi   30     90
2  Simran   22     75
```

📝 Hinglish: Har row me ek naya column Score add kar diya.

🚗 2. Add Column with Scalar Value

```
In [2]:
```

```
df['Country'] = "India"
print(df)
```

```
     Name  Age  Score Country
0    Amit   25     80   India
1    Ravi   30     90   India
2  Simran   22     75   India
```

📝 Hinglish: Har row ke liye Country = `India` add kar diya.

📊 3. Drop Single Column

```
In [5]:
```

```
print(df.drop("Age", axis=1))
```

```
     Name  Score Country
0    Amit     80   India
1    Ravi     90   India
2  Simran     75   India
```

📝 Hinglish: Age column remove kar diya.

⚽ 4. Drop Multiple Columns

```
In [14]:
```

```
print(df.drop(["Age", "Country"], axis=1))
```

```
     Name  Score
0    Amit     80
1    Ravi     90
2  Simran     75
```

📝 Hinglish: Ek saath multiple columns remove kar diye.

💰 5. Real-Life Example: Sales Data Column Engineering

```
In [17]:
```

```
sales = {
    "OrderID": [101, 102, 103],
    "Customer": ["Raj", "Simran", "Aman"],
    "Amount": [250, 500, 150]
}

df = pd.DataFrame(sales)
print(df)

# Add GST (18%)
df["GST"] = df["Amount"] * 0.18

# Drop Customer column
df2 = df.drop("Customer", axis=1)
print("\n", df2)
```

```
   OrderID Customer  Amount
0      101      Raj     250
1      102   Simran     500
2      103     Aman     150

    OrderID  Amount   GST
0      101     250  45.0
1      102     500  90.0
2      103     150  27.0
```

📝 Hinglish: Naya column GST calculate karke add kiya aur Customer column remove kar diya.

📌 (iv) What is Insert Column?

📏 Definition — Insert Column at Specific Position

📖 English:

In **Pandas**, you can **insert a column** at a specific position using **.insert()**.

👉 **df.insert(1, "new_col", values)** → inserts a new column named **new_col** at index position **1**.

🗣️ Hinglish:

DataFrame me agar **naya column kisi specific position** (1st, 2nd, 3rd...) pe dalna ho, to **.insert()** use hota hai.

👉 **df.insert(1, "new_col", values)** → 2nd column ke jagah naya column ghus gaya.

👉 Useful jab **column ka order maintain** karna ho (jaise report ke liye).

🔹 Syntax:

```
df.insert(loc, column, value, allow_duplicates=False)
```

- **Parameters**:

  👉 **loc** → Position (0-based index) jaha column insert karna hai

  👉 **column** → New column ka naam

  👉 **value** → Column ke values (list, scalar, Series, etc.)

  👉 **allow_duplicates** → Default False (same naam ka column allow nahi karega)

- **Returns** → Modified DataFrame with new column inserted

📌 Real-Life Examples of Insert Column

🍎 1. Insert Column at 1st Position

In [18]:

```
import pandas as pd

data = {"Name": ["Amit", "Ravi", "Simran"], "Age": [25, 30, 22]}
df = pd.DataFrame(data)

df.insert(0, "ID", [101, 102, 103])
print(df)
```

```
    ID    Name  Age
0  101    Amit   25
1  102    Ravi   30
2  103  Simran   22
```

📝 Hinglish: `ID` column sabse pehle insert ho gaya.

🚗 2. Insert Column in Middle

In [19]:

```
df.insert(1, "Score", [85, 90, 75])
print(df)
```

```
    ID  Score    Name  Age
0  101     85    Amit   25
1  102     90    Ravi   30
2  103     75  Simran   22
```

📝 Hinglish: `Score` column ko 2nd position (index=1) me insert kar diya.

📊 3. Insert Column with Constant Value

In [20]:

```
df.insert(2, "Country", "India")
print(df)
```

```
    ID  Score Country    Name  Age
0  101     85   India    Amit   25
1  102     90   India    Ravi   30
2  103     75   India  Simran   22
```

📝 Hinglish: Har row ke liye `Country = India` column insert ho gaya.

⚽ 4. Insert Column Using Series (Align by Index)

In [21]:

```
import pandas as pd

df2 = pd.DataFrame({"Name": ["Amit", "Ravi", "Simran"], "Age": [25, 30, 22]})
series = pd.Series([1000, 2000, 3000], index=[0, 1, 2])

df.insert(1, "Salary", series)
print(df)
```

```
    ID  Salary  Score Country    Name  Age
0  101    1000     85   India    Amit   25
1  102    2000     90   India    Ravi   30
2  103    3000     75   India  Simran   22
```

📝 Hinglish: `Salary` column Series se insert ho gaya, index ke basis pe align karke.

💰 5. Real-Life Example: Sales Data Column Insertion

In [23]:

```
sales = {
    "OrderID": [101, 102, 103],
    "Customer": ["Raj", "Simran", "Aman"],
    "Amount": [250, 500, 150]
}

df = pd.DataFrame(sales)

# Insert GST column at position 2
df.insert(2, "GST", df["Amount"] * 0.18)

print(df)
```

```
   OrderID Customer   GST  Amount
0      101      Raj  45.0     250
1      102   Simran  90.0     500
2      103     Aman  27.0     150
```

📝 Hinglish: `GST` column ko 3rd position (Amount ke baad) insert kar diya.

📌 (v) What is Replace Values?

📏 Definition — Replace specific values in a DataFrame

📖 English:

In **Pandas**, **.replace()** is used to change specific values inside a DataFrame or Series.

You can replace **single values**, **multiple values**, or even use **regex patterns**.

👉 **df.replace({1: "A", 2: "B"})** → replaces **1** with **"A"** and **2** with **"B"**.

🗣️ Hinglish:

Agar DataFrame ke andar **kuch values ko directly badalna** ho (jaise **0 ko NaN**, ya **1 ko "Yes"**), to **.replace()** ka use hota hai.

👉 **df.replace({1: "A", 2: "B"})** → 1 replace ho gaya "A" se aur 2 replace ho gaya "B" se.

👉 Useful jab **encoding**, **cleaning**, ya **transformation** kar rahe ho.

🔹 Syntax:

```
df.replace(to_replace, value=None, inplace=False,
regex=False)
```

- **Parameters**:

👉 **to_replace** → Value ya dictionary jo replace karna hai

👉 **value** → Naya value jo dalna hai

👉 **inplace** → True (modify original), False (return new DataFrame)

👉 **regex** → Pattern matching ke liye True

- **Returns** → New DataFrame (agar inplace=False hai)

📌 Real-Life Examples of Replace Values

🍎 1. Replace Single Value`m

In [25]:

```python
import pandas as pd

data = {"Name": ["Amit", "Ravi", "Simran"], "Age": [25, 30, 22]}
df = pd.DataFrame(data)

print(df.replace(25, 26))
```

```
     Name  Age
0    Amit   26
1    Ravi   30
2  Simran   22
```

📝 Hinglish: Age=25 ko 26 me replace kar diya.

🚗 2. Replace Multiple Values with Dictionary

In [26]:

```python
print(df.replace({"Amit":"Aman", 30:31}))
```

```
     Name  Age
0    Aman   25
1    Ravi   31
2  Simran   22
```

📝 Hinglish: "Amit" → "Aman", aur 30 → 31 replace ho gaya.

📊 3. Replace Using List

In [27]:

```
print(df.replace([22, 30], [23, 35]))
```

```
     Name  Age
0    Amit   25
1    Ravi   35
2  Simran   23
```

📝 Hinglish: 22 ko 23 aur 30 ko 35 se replace kar diya.

⚽ 4. Replace with Regex Pattern

In [28]:

```
df2 = pd.DataFrame({"City": ["New Delhi", "Delhi NCR", "Old Delhi"]})
print(df2.replace(to_replace="Delhi", value="DL", regex=True))
```

```
     City
0  New DL
1  DL NCR
2  Old DL
```

📝 Hinglish: "Delhi" word jitna bhi tha, sab "DL" me replace ho gaya.

💰 5. Real-Life Example: Cleaning Sales Data

In [29]:

```
sales = {
    "OrderID": [101, 102, 103],
    "Customer": ["Raj", "Simran", "Aman"],
    "Status": ["Pending", "Done", "Pending"]
}

df = pd.DataFrame(sales)

# Replace "Pending" with "In Progress"
df["Status"] = df["Status"].replace("Pending", "In Progress")

print(df)
```

```
   OrderID Customer       Status
0      101      Raj  In Progress
1      102   Simran         Done
2      103     Aman  In Progress
```

📝 Hinglish: Status column me "Pending" ko "In Progress" se replace kar diya.

📌 (vi) What is Map Values?

📏 Definition — Transform or map values in a Series

📖 English:

In Pandas, **.map()** is mainly used with a **Series** (single column). It helps to **transform or map values** of a column by using a **dictionary** or a **function**.
👉 **df["col"].map({1: "One", 2: "Two"})** → Maps **1 → "One"**, **2 → "Two"**.
👉 **df["col"].map(lambda x: x*2)** → Doubles each value.

🗣️ Hinglish:

.map() function ek column ke andar **values ko replace ya transform** karne ke kaam aata hai.
👉 **Dictionary se** → Direct mapping (e.g., **1 → "One"**).
👉 **Function se** → Har value pe calculation ya logic apply hota hai.

👉 Example: **df["col"].map(lambda x: x*2)** → Har number ko double kar diya.
⚡ Ye mostly use hota hai **data cleaning**, **encoding**, aur **feature engineering** ke liye.

🔹 Syntax:

```
Series.map(arg, na_action=None)
```

- **Parameters**:
  👉 **arg** → Dict, function, or Series (jis se mapping karni hai)
  👉 **na_action** → "ignore" set karoge to NaN pe operation skip karega

- **Returns** → A new **Series** with mapped values

📌 Real-Life Examples of Map Values

🍎 1. Map Values Using Dictionary

In [30]:

```python
import pandas as pd

df = pd.DataFrame({"Marks": [1, 2, 3, 1, 2]})

print(df["Marks"].map({1: "Fail", 2: "Pass", 3: "Topper"}))
```

```
0      Fail
1      Pass
2    Topper
3      Fail
4      Pass
Name: Marks, dtype: object
```

📝 Hinglish: Har number ko dictionary ke hisaab se label me convert kar diya.

🚗 2. Map Values Using Lambda Function

In [31]:

```
df = pd.DataFrame({"Age": [20, 25, 30]})

print(df["Age"].map(lambda x: x + 5))
```

```
0    25
1    30
2    35
Name: Age, dtype: int64
```

📝 Hinglish: Har age ke upar +5 kar diya.

📊 3. Map with Built-in Function

In [33]:

```
df = pd.DataFrame({"Names": ["amit", "ravi", "simran"]})

print(df["Names"].map(str.upper))
```

```
0      AMIT
1      RAVI
2    SIMRAN
Name: Names, dtype: object
```

📝 Hinglish: Sab names ko uppercase me convert kar diya.

⚽ 4. Map with Missing Values

In [34]:

```
df = pd.DataFrame({"Grade": [1, 2, None, 3]})

print(df["Grade"].map({1: "A", 2: "B", 3: "C"}))
```

```
0      A
1      B
2    NaN
3      C
Name: Grade, dtype: object
```

📝 Hinglish: 1,2,3 ko grade letters me map kar diya, aur NaN ko ignore kar diya.

💰 5. Real-Life Example: Customer Segmentation

In [35]:

```
customers = {
    "CustomerID": [101, 102, 103, 104],
    "Gender": ["M", "F", "M", "F"]
}

df = pd.DataFrame(customers)

df["Gender_Full"] = df["Gender"].map({"M": "Male", "F": "Female"})
print(df)
```

```
   CustomerID Gender Gender_Full
0         101      M        Male
1         102      F      Female
2         103      M        Male
3         104      F      Female
```

📝 Hinglish: "M" aur "F" ko full form me convert kar diya (Male/Female).

📌 (vii) What is Applymap?

📏 Definition — Apply a function element-wise across the entire DataFrame

📖 English:

In Pandas, **.applymap()** is used **only with DataFrames**. It applies a given function **element-wise** (har cell individually).

👉 **df.applymap(str.upper)** → Converts every string to uppercase.

👉 **df.applymap(lambda x: x*2)** → Doubles every numeric value.

⚡ Useful jab aapko **poore DataFrame** me ek hi operation lagana ho.

🗣️ Hinglish:

.applymap() function DataFrame ke andar **har ek cell pe function apply** karta hai.
Matlab column-wise ya row-wise nahi, balki element by element kaam karta hai.

👉 Example: **df.applymap(str.upper)** → Sabhi strings ko uppercase kar diya.

👉 Example: **df.applymap(lambda x: x*2)** → Har number ko double kar diya.

⚡ Mostly use hota hai **string transformations**, **math operations**, ya **data cleaning** ke liye.

🔹 Syntax:

```
DataFrame.applymap(func)
```

- **Parameters**:

  👉 **func** → Function (lambda, built-in, ya custom function) jo har element pe apply hoga

- **Returns** → A new **DataFrame** with transformed values

📌 Real-Life Examples of Applymap

🍎 1. Applymap with Lambda Function

In [36]:

```
import pandas as pd

df = pd.DataFrame({"A": [1, 2, 3], "B": [4, 5, 6]})

print(df.applymap(lambda x: x * 10))
```

```
    A   B
0  10  40
1  20  50
2  30  60
C:\Users\gkuir\AppData\Local\Temp\ipykernel_15588\416796933.py:5: FutureWarning: DataFrame.applymap has been deprecated. Use DataFrame.map instead.
  print(df.applymap(lambda x: x * 10))
```

📝 Hinglish: Har number ko 10 se multiply kar diya.

🚗 2. Applymap with Built-in String Function

In [37]:

```
df = pd.DataFrame({"Name": ["amit", "ravi"], "City": ["delhi", "mumbai"]})

print(df.applymap(str.upper))
```

```
   Name    City
0  AMIT   DELHI
1  RAVI  MUMBAI
C:\Users\gkuir\AppData\Local\Temp\ipykernel_15588\3960684099.py:3: FutureWarning: DataFrame.applymap has been deprecated. Use DataFrame.map instead.
  print(df.applymap(str.upper))
```

📝 Hinglish: Har string ko uppercase me convert kar diya.

📊 3. Applymap with Conditional Function

In [38]:

```
df = pd.DataFrame({"Score1": [45, 80], "Score2": [30, 90]})

print(df.applymap(lambda x: "Pass" if x >= 40 else "Fail"))
```

```
  Score1 Score2
0   Pass   Fail
1   Pass   Pass
C:\Users\gkuir\AppData\Local\Temp\ipykernel_15588\2664217885.py:3: FutureWarning: DataFrame.applymap has been deprecated. Use DataFrame.map instead.
```

```
  print(df.applymap(lambda x: "Pass" if x >= 40 else "Fail"))
```

📝 Hinglish: Har score ko condition ke hisaab se "Pass" ya "Fail" me badal diya.

⚽ 4. Applymap on Mixed Data

In [39]:

```
df = pd.DataFrame({"A": [1, "hello"], "B": [3.5, "world"]})

print(df.applymap(str))
```

```
       A      B
0      1    3.5
1  hello  world
C:\Users\gkuir\AppData\Local\Temp\ipykernel_15588\1901183044.py:3: FutureWarning: DataFr
ame.applymap has been deprecated. Use DataFrame.map instead.
  print(df.applymap(str))
```

📝 Hinglish: Sabhi values ko string me convert kar diya.

💰 5. Real-Life Example: Currency Formatting

In [42]:

```
sales = {
    "Product": ["Pen", "Book", "Pencil"],
    "Price": [10, 50, 5]
}

df = pd.DataFrame(sales)

df_formatted = df.applymap(lambda x: f"${x}" if isinstance(x, int) else x)
print(df_formatted)
```

```
  Product Price
0     Pen   $10
1    Book   $50
2  Pencil    $5
C:\Users\gkuir\AppData\Local\Temp\ipykernel_15588\1937359901.py:8: FutureWarning: DataFr
ame.applymap has been deprecated. Use DataFrame.map instead.
  df_formatted = df.applymap(lambda x: f"${x}" if isinstance(x, int) else x)
```

📝 Hinglish: Price column ke numbers ko currency format me convert kar diya (₹/$ ke saath).

7. 📊 Grouping & Aggregations – Summarize Your Data

Operation	Definition	Syntax / Example

Group By	Group data based on column(s).	`df.groupby("col")`
		`df.groupby(["col1","col2`

Aggregate Functions	Apply summary stats like sum, mean, count.	`df.groupby("col")["sales"].sum()`
		`df.groupby("col")`

Operation	Definition	Syntax / Example

		`["marks"].mean()`
		`df.groupby("col").count(`

Multiple Aggregations	Apply multiple aggregations using `agg()`.	`df.groupby("col")["marks"].agg(["sum","me`
Multi-level GroupBy	Group data on multiple columns (hierarchical).	`df.groupby(["region","ci["sales"].sum()`
Pivot Table	Summarize data like Excel Pivot Table.	`df.pivot_table(values="s index="region", columns="year", aggfunc="sum")`

📌 (i) What is GroupBy?

📏 **Definition — Group data based on one or more columns**

📖 English:

In **Pandas**, **.groupby()** groups data using one or more column values. After grouping, you can apply **aggregation functions** like sum, mean, or count.

👉 **df.groupby("col").sum()** → Groups by a single column and calculates sum.

👉 **df.groupby(["col1", "col2"]).mean()** → Groups by multiple columns and calculates mean.

🗣️ Hinglish:

.groupby() ka use tab hota hai jab hume **data ko categories ke hisaab se group** karna ho aur fir uspe calculation karni ho.

👉 Example: **df.groupby("City").sum()** → Har city ka total nikal diya.

👉 Example: **df.groupby(["Gender", "Class"]).mean()** → Gender + Class ke basis pe average nikal diya.

⚡ Useful for **sales analysis, student marks analysis,** aur summary statistics.

- 🔹 Syntax:

```
DataFrame.groupby(by, axis=0).agg(func)
```

- **Parameters**:

  👉 **by** → Column name(s) jinke basis pe grouping karni hai

  👉 **axis** → 0 = rows (default), 1 = columns

  👉 **agg(func)** → Aggregation function (sum, mean, count, max, min, etc.)

- **Returns** → Grouped **DataFrame** or **Series**

📌 Real-Life Examples of GroupBy

🍎 1. GroupBy Single Column

In [45]:

```
import pandas as pd

data = {"City": ["Delhi", "Delhi", "Mumbai", "Mumbai", "Chennai"],
        "Sales": [100, 200, 150, 250, 300]}

df = pd.DataFrame(data)

print(df.groupby("City").sum("Sales"))
```

```
         Sales
City
Chennai    300
Delhi      300
Mumbai     400
```

📝 Hinglish: Har city ka total sales nikal liya.

🚗 2. GroupBy Multiple Columns

In [50]:

```
data = {"Class": ["A", "A", "B", "B"],
        "Gender": ["M", "F", "M", "F"],
        "Marks": [80, 90, 70, 60]}

df = pd.DataFrame(data)

print(df.groupby(["Class", "Gender"]).mean("Marks"))

print("\n", df.groupby(["Class", "Gender"])["Marks"].mean())
```

```
              Marks
Class Gender
A     F        90.0
      M        80.0
B     F        60.0
      M        70.0
```

```
 Class  Gender
A      F         90.0
       M         80.0
B      F         60.0
       M         70.0
Name: Marks, dtype: float64
```

📝 Hinglish: Class aur Gender ke hisaab se average marks calculate ho gaye.

📊 3. GroupBy with Multiple Aggregations

In [51]:

```
data = {"Team": ["A", "A", "B", "B", "C"],
        "Points": [10, 20, 15, 25, 30],
        "Games": [2, 3, 2, 4, 5]}

df = pd.DataFrame(data)

print(df.groupby("Team").agg({"Points":"sum", "Games":"mean"}))
```

```
      Points  Games
Team
A         30    2.5
B         40    3.0
C         30    5.0
```

📝 Hinglish: Har team ke liye Points ka total aur Games ka average nikal liya.

⚽ 4. GroupBy with Custom Function

In [52]:

```
data = {"Category": ["X", "X", "Y", "Y"],
        "Value": [5, 15, 20, 25]}

df = pd.DataFrame(data)

print(df.groupby("Category")["Value"].apply(lambda x: x.max() - x.min()))
```

```
Category
X    10
Y     5
Name: Value, dtype: int64
```

📝 Hinglish: Har category ke andar max - min (range) nikal liya.

💰 5. Real-Life Example: Customer Sales Analysis

In [53]:

```
sales = {
    "Customer": ["C1", "C2", "C1", "C3", "C2"],
    "Amount": [200, 150, 300, 400, 250]
}

df = pd.DataFrame(sales)

print(df.groupby("Customer")["Amount"].sum())
```

```
Customer
C1    500
C2    400
C3    400
Name: Amount, dtype: int64
```

📝 Hinglish: Har customer ka total sales calculate ho gaya.

📌 (ii) What are Aggregate Functions?

📏 **Definition — Apply summary statistics like sum, mean, count, max, min etc. on DataFrame or grouped data**

📖 English:

In **Pandas**, aggregate functions are built-in methods used to calculate **summary statistics** over a dataset.
They return a **single value** for a column or group of rows (like total sales, average marks, or row counts).
👉 **df["Sales"].sum()** → Returns total sales.
👉 **df["Marks"].mean()** → Returns average marks.
👉 **df.count()** → Counts non-NaN values in each column.

🗣️ Hinglish:

Aggregate functions ka use tab hota hai jab hume **data ka ek summary number** chahiye hota hai.
Matlab ek column ya group ka **sum, average, count, max, ya min** nikalna.

👉 Example: **df["Sales"].sum()** → Total sales nikal gaya.
👉 Example: **df.groupby("City")["Sales"].sum()** → Har city ka sales total mil gaya.
⚡ Useful for **reporting, analysis, aur data summarization**.

🔹 Syntax:

```
DataFrame.agg(func)
DataFrame.groupby(col).agg(func)
```

- **Parameters**:

  👉 **func** → Aggregation function (sum, mean, count, max, min, std, median, etc.)

  👉 **by (optional)** → Grouping columns (agar groupby ke saath use kar rahe ho)

- **Returns** → Scalar value (single number) ya **DataFrame** (groupby ke saath)

📌 Real-Life Examples of Aggregate Functions

🍎 1. Aggregate: Sum

In [54]:

```python
import pandas as pd

df = pd.DataFrame({"City": ["Delhi", "Delhi", "Mumbai"],
                   "Sales": [100, 200, 300]})

print(df["Sales"].sum())
```

```
600
```

📝 Hinglish: Total sales ka sum nikal diya (100+200+300 = 600).

🚗 2. Aggregate: Mean (Average)

In [55]:

```python
df = pd.DataFrame({"Student": ["A", "B", "C"],
                   "Marks": [80, 90, 70]})

print(df["Marks"].mean())
```

```
80.0
```

📝 Hinglish: Students ke marks ka average nikal liya.

📊 3. Aggregate: Count

In [56]:

```python
df = pd.DataFrame({"City": ["Delhi", "Mumbai", "Delhi"],
                   "Sales": [100, 150, 200]})


print(df["City"].count())
```

```
3
```

📝 Hinglish: Column me kitni values (NaN ke alawa) hain, uska count kar diya.

⚽ 4. Aggregate with GroupBy

In [57]:

```python
df = pd.DataFrame({"City": ["Delhi", "Delhi", "Mumbai"],
                   "Sales": [100, 200, 300]})
```

```
print(df.groupby("City")["Sales"].sum())
```

```
City
Delhi     300
Mumbai    300
Name: Sales, dtype: int64
```

📝 Hinglish: Har city ke liye total sales nikal diya.

💰 5. Real-Life Example: Employee Salary Analysis

```
In [65]:
salary = {
    "Dept": ["IT", "IT", "HR", "HR", "Finance"],
    "Salary": [50000, 60000, 40000, 45000, 70000]
}


df = pd.DataFrame(salary)

print(df.groupby("Dept")["Salary"].mean().round(0).astype(int))
```

```
Dept
Finance    70000
HR         42500
IT         55000
Name: Salary, dtype: int64
```

📝 Hinglish: Har department ke liye average salary nikal liya.

📌 (iii) What are Multiple Aggregations?

📏 Definition — Apply multiple aggregation functions on one or more columns using .agg() after optional groupby()

📖 English:

Multiple aggregations allow you to calculate **different summary statistics** on the same column or multiple columns simultaneously. In Pandas, use **.agg()** to compute sum, mean, max, min, count, etc. in a single command.

👉 **df.groupby("col")["marks"].agg(["sum", "mean", "max"])** → Returns sum, mean, and max for the Marks column.

👉 **df.agg(["sum", "mean"])** → Calculates sum and mean for all numeric columns in the DataFrame.

🗣️ Hinglish:

Multiple aggregations ka matlab hai ek column ya multiple columns pe ek saath alag-alag summary statistics nikalna.
Jaise ek hi command me total, average, aur maximum nikalna.

👉 Example: **df.groupby("City")["Sales"].agg(["sum","mean","max"])** → Har city ka total, average aur max sales nikal gaya.
👉 Example: **df.agg(["sum","mean"])** → Sare numeric columns ka total aur average ek saath mil gaya.
⚡ Mostly use hota hai **reporting, analysis, aur data summary** me.

🔹 Syntax:

```
DataFrame.agg(func_list)
DataFrame.groupby(col).agg(func_list_or_dict)
```

- **Parameters**:
 👉 **func_list** → List of function names or callable functions (e.g., ["sum","mean","max"])
 👉 **func_dict** → Dictionary {column_name: [func1, func2]} for different functions on different columns
 👉 **by** → Grouping columns (agar groupby ke saath use kar rahe ho)

- **Returns** → A new **DataFrame** with multiple aggregated values

📌 Real-Life Examples of Multiple Aggregations

🍎 1. Multiple Aggregations on Single Column

In [66]:

```python
import pandas as pd

df = pd.DataFrame({"City": ["Delhi", "Delhi", "Mumbai"],
                   "Sales": [100, 200, 300]})

print(df.groupby("City")["Sales"].agg(["sum", "mean", "max"]))
```

```
        sum   mean  max
City
Delhi   300  150.0  200
Mumbai  300  300.0  300
```

📝 Hinglish: Har city ke liye **total, average aur maximum sales** ek saath nikal diya.

🚗 2. Multiple Aggregations on Entire DataFrame

In [67]:

```
df = pd.DataFrame({"Marks": [80, 90, 70],
                   "Bonus": [10, 15, 5]})

print(df.agg(["sum", "mean"]))
```

```
      Marks  Bonus
sum   240.0   30.0
mean   80.0   10.0
```

📝 Hinglish: Marks aur Bonus dono columns ke **total aur average** ek saath calculate ho gaye.

📊 3. Different Functions on Different Columns

In [68]:

```
df = pd.DataFrame({"Team": ["A","A","B"], "Points": [10,20,15], "Games": [2,3,2]})

print(df.groupby("Team").agg({"Points":"sum", "Games":"mean"}))
```

```
      Points  Games
Team
A         30    2.5
B         15    2.0
```

📝 Hinglish: Team ke liye **Points ka total** aur **Games ka average** alag-alag apply kiya.

⚽ 4. Custom Aggregations

In [70]:

```
df = pd.DataFrame({"Category": ["X","X","Y"], "Value": [5,15,20]})

print(df.groupby("Category")["Value"].agg([lambda x: x.max() - x.min()]))
```

```
          <lambda>
Category
X               10
Y                0
```

📝 Hinglish: Har category ke liye **max-min difference** calculate kar diya.

💰 5. Real-Life Example: Employee Salary Analysis

In [71]:

```
salary = {
    "Dept": ["IT","IT","HR","HR","Finance"],
    "Salary": [50000,60000,40000,45000,70000]
}

df = pd.DataFrame(salary)

print(df.groupby("Dept")["Salary"].agg(["sum","mean","max"]))
```

```
            sum     mean    max
Dept
Finance   70000  70000.0  70000
HR        85000  42500.0  45000
IT       110000  55000.0  60000
```

📝 Hinglish: Har department ke liye **total, average aur maximum salary** ek saath nikal diya.

📌 (iv) What is Multi-level GroupBy?

📏 **Definition — Group data based on multiple columns (hierarchical grouping) and optionally apply aggregation functions**

📖 English:

Multi-level GroupBy means grouping data using two or more columns simultaneously (hierarchical grouping). You can then apply aggregation functions like sum, mean, count, max, min, etc.
👉 **df.groupby(["region","city"])["sales"].sum()** → Groups by region first, then by city, and calculates total sales.
👉 **df.groupby(["Dept","Team"]).agg({"Salary":["sum","mean"]})** → Calculates total and average salary by Department and Team.

🗣️ Hinglish:

Multi-level GroupBy tab use hota hai jab hume data ko **do ya do se zyada dimensions** me analyze karna ho.
Jaise region → city ke basis pe sales dekhna, ya department → team ke basis pe salary summary nikalna.

👉 Example: **df.groupby(["Region","City"])["Sales"].sum()** → Region aur city dono ke hisaab se total sales.
👉 Example: **df.groupby(["Dept","Team"]).agg(["sum","mean"])** → Har Dept + Team combination ke liye total aur average salary.
⚡ Mostly use hota hai **hierarchical reporting, pivot analysis, aur multi-dimensional aggregation** me.

🔹 Syntax:

```
DataFrame.groupby([col1, col2, ...])
[target_col].agg(func_list_or_dict)
```

- **Parameters**:

👉 **[col1, col2,...]** → Multiple columns to group by

👉 **target_col** → Column(s) on which aggregation is applied

👉 **agg(func_list_or_dict)** → List of functions or dictionary for multiple aggregations

- **Returns** → Multi-level indexed **DataFrame** with aggregated values

📌 Real-Life Examples of Multi-level GroupBy

🍎 1. GroupBy Two Columns

In [73]:

```
import pandas as pd

df = pd.DataFrame({
    "Region": ["East","East","West","West","East"],
    "City": ["Delhi","Lucknow","Mumbai","Pune","Delhi"],
    "Sales": [100,150,200,250,300]
})

print(df.groupby(["Region", "City"])["Sales"].sum())
```

```
Region  City
East    Delhi      400
        Lucknow    150
West    Mumbai     200
        Pune       250
Name: Sales, dtype: int64
```

📝 Hinglish: Region → City ke basis pe total sales calculate ho gaya.

🚗 2. Multi-level GroupBy with Multiple Aggregations

In [75]:

```
df = pd.DataFrame({
    "Dept": ["IT","IT","HR","HR","Finance"],
    "Team": ["A","B","A","B","C"],
    "Salary": [50000,60000,40000,45000,70000]
})

print(df.groupby(["Dept", "Team"])["Salary"].agg(["sum", "mean", "max"]).round(0).astype
```

```
                sum   mean    max
Dept    Team
Finance C     70000  70000  70000
HR      A     40000  40000  40000
        B     45000  45000  45000
```

```
IT      A     50000  50000  50000
        B     60000  60000  60000
```

📝 Hinglish: Department aur Team ke liye salary ka total, average aur maximum ek saath nikal diya.

📊 3. Different Functions on Multiple Columns

In [76]:

```
df = pd.DataFrame({
    "Region": ["North","North","South","South"],
    "City": ["Delhi","Delhi","Chennai","Chennai"],
    "Sales": [100,200,150,250],
    "Profit": [10,20,15,25]
})

print(df.groupby(["Region", "City"]).agg({"Sales":"sum", "Profit":"mean"}))
```

```
                Sales  Profit
Region City
North  Delhi      300    15.0
South  Chennai    400    20.0
```

📝 Hinglish: Region + City ke combination ke liye **Sales ka total aur Profit ka average** calculate kiya.

⚽ 4. Custom Aggregations with Lambda

In [77]:

```
df = pd.DataFrame({
    "Category": ["X","X","Y","Y"],
    "Type": ["A","B","A","B"],
    "Value": [5,15,20,25]
})

print(df.groupby(["Category","Type"])["Value"].agg([lambda x: x.max()-x.min()]))
```

```
               <lambda>
Category Type
X        A            0
         B            0
Y        A            0
         B            0
```

In [79]:

```
df = pd.DataFrame({
    "Category": ["X","X","Y","Y"],
    "Type": ["A","A","B","B"],
    "Value": [5,15,20,25]
})

print(df.groupby(["Category","Type"])["Value"].agg([lambda x: x.max()-x.min()]))
```

```
               <lambda>
Category Type
X        A           10
Y        B            5
```

📝 Hinglish: Har Category + Type ke liye **Value ka range (max-min)** calculate kiya.

💰 5. Real-Life Example: Sales Performance Analysis

In [80]:

```
sales = {
    "Region": ["East","East","West","West","East"],
    "City": ["Delhi","Lucknow","Mumbai","Pune","Delhi"],
    "Amount": [200,150,300,400,250]
}

df = pd.DataFrame(sales)

print(df.groupby(["Region","City"])["Amount"].agg(["sum","mean","max"]))
```

```
                sum   mean  max
Region City
East   Delhi    450  225.0  250
       Lucknow  150  150.0  150
West   Mumbai   300  300.0  300
       Pune     400  400.0  400
```

📝 Hinglish: Region aur City combination ke liye **total, average aur maximum sales** nikal diya.

📌 (v) What is Pivot Table?

📏 Definition — Summarize and analyze data in a table like Excel Pivot Table, aggregating values across rows and columns

📖 English:

Pivot Table is a **Pandas** function used to summarize data. It aggregates values across rows (index) and columns, similar to Excel Pivot Table.

You can apply aggregation functions like sum, mean, count, etc.

👉 **df.pivot_table(values="sales", index="region", columns="year", aggfunc="sum")** → Calculates total sales for each region and year.

👉 **df.pivot_table(values=["Sales","Profit"], index="Dept", columns="Team", aggfunc=["sum","mean"])** → Multiple metrics and aggregations in one table.

🗣️ Hinglish:

Pivot Table tab use hota hai jab hume **cross-tab analysis** karna ho, jaise region → year → sales summary, ya department → team → salary summary.

👉 Example: **df.pivot_table(values="Amount", index="City", columns="Month", aggfunc="sum")** → City aur Month ke hisaab se total amount.

👉 Example: **df.pivot_table(values="Marks", index="Class", columns="Gender",**

aggfunc="mean") → Class aur Gender combination ke liye average marks.

⚡ Mostly use hota hai **reporting, business analysis, aur quick summarization** me.

🔹 Syntax:

```
DataFrame.pivot_table(
    values=column_to_aggregate,
    index=row_grouping_columns,
    columns=column_grouping_columns,
    aggfunc="sum" or ["sum","mean"],
    fill_value=None,
    margins=False
)
```

- **Parameters**:

👉 **values** → Column(s) jisko summarize karna hai

👉 **index** → Row ke liye grouping columns

👉 **columns** → Column-wise grouping

👉 **aggfunc** → Aggregation function(s) like sum, mean, count, max, min

👉 **fill_value** → Missing values ke liye fill karo

👉 **margins** → True set karoge to row aur column totals nikalega

- **Returns** → Pivoted **DataFrame**

📌 Real-Life Examples of Pivot Table

🍎 1. Simple Pivot Table

In [82]:

```python
import pandas as pd

df = pd.DataFrame({
    "Region": ["East","East","West","West","East"],
    "Year": [2020,2021,2020,2021,2020],
    "Sales": [100,150,200,250,300]
})

print(df.pivot_table(values="Sales", index="Region", columns="Year", aggfunc="sum"))
```

```
Year    2020  2021
Region
East     400   150
West     200   250
```

📝 Hinglish: Region aur Year ke combination ke liye total sales nikal diya.

🚗 2. Pivot Table with Multiple Aggregations

In [93]:

```
df = pd.DataFrame({
    "Dept": ["IT","IT","HR","HR","Finance"],
    "Team": ["A","B","A","B","C"],
    "Salary": [50000,60000,40000,45000,70000]
})

print(df.pivot_table(values="Salary", index="Dept", columns="Team", aggfunc=["sum","mean
```

```
             sum                          mean
Team           A        B        C        A        B        C
Dept
Finance      NaN      NaN  70000.0      NaN      NaN  70000.0
HR       40000.0  45000.0      NaN  40000.0  45000.0      NaN
IT       50000.0  60000.0      NaN  50000.0  60000.0      NaN
```

In [94]:

```
import pandas as pd

df = pd.DataFrame({
    "Dept": ["IT","IT","HR","HR","Finance"],
    "Team": ["A","B","A","B","C"],
    "Salary": [50000,60000,40000,45000,70000]
})

# Pivot table
pt = df.pivot_table(values="Salary", index="Dept", columns="Team", aggfunc=["sum","mean"

# Fill NaN with 0 (ya koi suitable value), then round and convert to int
pt = pt.fillna(0).round(0).astype(int)

print(pt)
```

```
           sum                  mean
Team         A      B      C      A      B      C
Dept
Finance      0      0  70000      0      0  70000
HR       40000  45000      0  40000  45000      0
IT       50000  60000      0  50000  60000      0
```

In [86]:

```
df = pd.DataFrame({
    "Dept": ["IT","IT","HR","HR","Finance"],
    "Team": ["A","B","A","B","C"],
    "Salary": [50000,60000,40000,45000,70000]
})

pt = df.pivot_table(values="Salary", index=["Dept", "Team"], aggfunc=["sum", "mean"])

# Flatten columns
pt.columns = ["Total Salary" if col[0] == "sum" else 'Average Salary' for col in pt.colu

print(pt)
```

```
              Total Salary  Average Salary
Dept    Team
Finance C            70000         70000.0
HR      A            40000         40000.0
        B            45000         45000.0
IT      A            50000         50000.0
        B            60000         60000.0
```

📝 Hinglish: Department aur Team combination ke liye **total aur average salary** ek saath calculate ho gaya.

📊 3. Pivot Table with Multiple Values

In [95]:

```
df = pd.DataFrame({
    "Region": ["North","North","South","South"],
    "City": ["Delhi","Delhi","Chennai","Chennai"],
    "Sales": [100,200,150,250],
    "Profit": [10,20,15,25]
})

print(df.pivot_table(values=["Sales", "Profit"], index="Region", columns="City", aggfunc
```

```
        Profit         Sales
City   Chennai Delhi Chennai  Delhi
Region
North      NaN  30.0     NaN  300.0
South     40.0   NaN   400.0    NaN
```

📝 Hinglish: Region aur City ke combination ke liye **Sales aur Profit ka total** ek saath.

⚽ 4. Pivot Table with Fill Value and Margins``

In [102]:

```
df = pd.DataFrame({
    "Region": ["East","East","West","West","East"],
    "Year": [2020,2021,2020,2021,2020],
    "Sales": [100,150,200,250,300]
})

print(df.pivot_table(values="Sales", index="Region", columns="Year", aggfunc="sum", fill
```

```
Year    2020  2021   All
Region
East     400   150   550
West     200   250   450
All      600   400  1000
```

📝 Hinglish: Missing combinations ko 0 se fill kiya aur **row/column totals** bhi nikal diye.

💰 5. Real-Life Example: Exam Scores Analysis

In [103]:

```
df = pd.DataFrame({
    "Class": ["A","A","B","B"],
    "Gender": ["M","F","M","F"],
    "Marks": [80,90,70,60]
})
```

```
print(df.pivot_table(values="Marks", index="Class", columns="Gender", aggfunc="mean"))
```

```
Gender     F     M
Class
A       90.0  80.0
B       60.0  70.0
```

📝 Hinglish: Class aur Gender combination ke liye **average marks** calculate kiya.

📌 GroupBy vs Pivot Table in Pandas

Feature	GroupBy	Pivot Table
Purpose	**Data ko group karke aggregation karna**	**Excel-style cross-tab summarization**
Input	**Column(s) for grouping**	**Index (rows), Columns, Values**
Aggregation	**.agg(), .sum(), .mean() etc.**	**aggfunc="sum" / "mean" / list of functions**
Output	**Series ya DataFrame** (usually hierarchical index if multi-level)	**DataFrame with row and column structure**, easy to read
Flexibility	**Highly flexible** for custom aggregations, lambda functions	**Mostly used for quick summaries**, simple aggregations
Missing Data Handling	**Needs .fillna() manually**	**fill_value parameter available**
Multi-level Grouping	**Multi-level index** with .groupby([col1, col2])	**Multi-level rows and columns** possible, visually like a matrix
When to Use	**Custom stats, multiple functions per column, feature engineering**	**Easy reporting, pivot-style tables, cross-tab analysis**
Complexity	**Slightly more coding**, more control	**Less code, more visual**, Excel-like
Performance	**Usually faster** for large datasets	**Slightly slower** for huge datasets because of pivot reshaping

8. 🔗 Merging, Joining & Concatenation – Combine Data

Operation	Definition	Syntax / Example

Concatenation	Stack DataFrames vertically (rows) or horizontally (columns).	`pd.concat([df1, df2])`
		`pd.concat([df1, df2], axis=1)`

Merge	SQL-style joins based on common columns or keys.	`pd.merge(df1, df2, on="id")`
		`pd.merge(df1, df2, how="inner")`
		`pd.merge(df1, df2, how="outer")`
		`pd.merge(df1, df2, how="left")`
		`pd.merge(df1, df2, how="right")`

Join	Join DataFrames using their index or key column.	`df1.join(df2)`
		`df1.join(df2, how="outer")`

📌 (i) What is Concatenation?

📏 **Definition — Concatenation ka matlab hai do ya zyada DataFrames ko stack karna ek saath — vertically (rows ke niche rows) ya horizontally (columns ke side me columns).**

📖 English:

Concatenation is used to combine multiple DataFrames into one.
By default, Pandas row-wise (axis=0) stack karta hai, matlab ek DataFrame ke neeche dusra add ho jata hai. Agar **axis=1** use kare to side-by-side (column-wise) merge ho jaata hai.

👉 **pd.concat([df1, df2])** → Row-wise (default).
👉 **pd.concat([df1, df2], axis=1)** → Column-wise.

🗣️ Hinglish:

Concatenation matlab DataFrames ko jodna. Jaise **LEGO blocks** — upar neeche lagao to rows add ho jaate hain, side-by-side lagao to columns add ho jaate hain.

👉 Example: **pd.concat([df1, df2])** → df1 ke rows ke neeche df2 ke rows chipak gaye.

👉 Example: **pd.concat([df1, df2], axis=1)** → df1 ke columns ke side me df2 ke columns aa gaye.

⚡ Ye useful hai jab data alag-alag file/dataset me ho aur usko ek jagah combine karna ho.

🔹 Syntax:

```
pd.concat(
    objs=[df1, df2],
    axis=0,
    join="outer",
    ignore_index=False,
    keys=None
)
```

- **Parameters**:

👉 **objs** → List of DataFrames to concatenate

👉 **axis** → 0 (rows/vertical) or 1 (columns/horizontal)

👉 **join** → "outer" (union) or "inner" (intersection) of indexes

👉 **ignore_index** → True karne pe naya index generate hoga

👉 **keys** → MultiIndex create karne ke liye labels

- **Returns** → New concatenated **DataFrame**

📌 Real-Life Examples of Concatenation

🍎 1. Row-wise Concatenation (Default)

In [2]:

```
import pandas as pd

df1 = pd.DataFrame({"A":[1,2], "B":[3,4]})
df2 = pd.DataFrame({"A":[5,6], "B":[7,8]})

print(pd.concat([df1, df2]))
```

```
   A  B
0  1  3
1  2  4
0  5  7
1  6  8
```

📝 Hinglish: df1 aur df2 ke rows ek ke neeche ek lag gaye.

🚗 2. Column-wise Concatenation

In [4]:

```
df1 = pd.DataFrame({"A":[1,2]})
df2 = pd.DataFrame({"B":[3,4]})

print(pd.concat([df1, df2], axis=1))
```

```
   A  B
0  1  3
1  2  4
```

📝 Hinglish: df1 aur df2 side-by-side combine ho gaye.

📊 3. Ignore Index in Row Concatenation

In [5]:

```
df1 = pd.DataFrame({"A":[10,20]})
df2 = pd.DataFrame({"A":[30,40]})

print(pd.concat([df1, df2], ignore_index=True))
```

```
    A
0  10
1  20
2  30
3  40
```

📝 Hinglish: Index reset karke 0,1,2,3 se naya DataFrame ban gaya.

⚽ 4. Concatenation with Different Columns (Outer Join)

In [6]:

```
df1 = pd.DataFrame({"A":[1,2], "B":[3,4]})
df2 = pd.DataFrame({"A":[5,6], "C":[7,8]})

print(pd.concat([df1, df2]))
```

```
   A    B    C
0  1  3.0  NaN
1  2  4.0  NaN
0  5  NaN  7.0
1  6  NaN  8.0
```

📝 Hinglish: Dono DataFrames ke columns merge ho gaye, missing jagah NaN aa gaya.

💰 5. Real-Life Example: Monthly Sales Data

In [7]:

```
jan = pd.DataFrame({"Month":["Jan","Jan"], "Sales":[100,200]})
feb = pd.DataFrame({"Month":["Feb","Feb"], "Sales":[150,250]})

print(pd.concat([jan, feb], ignore_index=True))
```

```
  Month  Sales
0   Jan    100
1   Jan    200
2   Feb    150
3   Feb    250
```

📝 Hinglish: January aur February ka sales data combine ho gaya ek DataFrame me.

📌 (ii) What is Merge?

📏 **Definition — Merge ka matlab hai do DataFrames ko common column ya index ke base par jodna, bilkul SQL join ki tarah (inner, outer, left, right).**

📖 English:

Merge is used to combine two DataFrames based on one or more common columns (keys). Ye SQL-style joins ki tarah kaam karta hai — jaise **inner join**, **left join**, **right join**, ya **outer join**.

👉 **pd.merge(df1, df2, on="id")** → Common "id" column ke base pe merge karega.
👉 **pd.merge(df1, df2, how="outer")** → Dono DataFrames ka full outer join karega.

🗣️ Hinglish:

Merge matlab DataFrames ko common key ke basis par jodna, jaise SQL ke joins hote hain. Default **inner join** hota hai, lekin tum **how** parameter se left, right, ya outer join kar sakte ho.

👉 Example: **pd.merge(df1, df2, on="id")** → Sirf matching "id" ke rows ko merge karega.
👉 Example: **pd.merge(df1, df2, how="left")** → Left DataFrame ke saare rows rakhega aur matching right ke add karega.

⚡ Ye useful hai jab data alag-alag tables me ho aur unko ek saath analysis ke liye jodna ho.

🔹 Syntax:

```
pd.merge(
    left=df1,
    right=df2,
    how="inner",
    on=None,
    left_on=None,
    right_on=None
)
```

- **Parameters:**
  👉 **left, right** → DataFrames to merge
  👉 **how** → Join type: "inner" (default), "outer", "left", "right"
  👉 **on** → Common column(s) to merge on
  👉 **left_on, right_on** → Agar column names alag hain dono DataFrames me
  👉 **suffixes** → Same column names ke liye suffix add karne ke liye

- **Returns** → New merged **DataFrame**

📌 Real-Life Examples of Merge

🍎 1. Inner Join (Default)

In [8]:

```
import pandas as pd

df1 = pd.DataFrame({"id":[1,2,3], "Name":["A","B","C"]})
df2 = pd.DataFrame({"id":[2,3,4], "Age":[20,25,30]})

print(pd.merge(df1, df2, on='id'))
```

```
   id Name  Age
0   2    B   20
1   3    C   25
```

📝 Hinglish: Sirf id=2 aur id=3 dono me match kiya.

🚗 2. Left Join

print(pd.merge(df1, df2, on='id', how='left'))

📝 Hinglish: Left DataFrame ke saare rows aaye, matching Age add hua, baki jagah NaN.

📊 3. Right Join

In [11]:

```
print(pd.merge(df1, df2, on='id', how="right"))
```

```
   id Name  Age
0   2    B   20
1   3    C   25
2   4  NaN   30
```

📝 Hinglish: Right DataFrame ke saare rows aaye, Name missing jagah NaN.

⚽ 4. Outer Join

In [12]:

```
print(pd.merge(df1, df2, on='id', how='outer'))
```

```
   id Name   Age
0   1    A   NaN
```

```
1    2    B  20.0
2    3    C  25.0
3    4  NaN  30.0
```

📝 Hinglish: Dono DataFrames ke saare ids aa gaye, missing values NaN.

💰 5. Real-Life Example: Employee Department Data

In [13]:

```python
emp = pd.DataFrame({"EmpID":[1,2,3], "Name":["Raj","Amit","Priya"]})
dept = pd.DataFrame({"EmpID":[2,3,4], "Dept":["HR","IT","Finance"]})

print(pd.merge(emp, dept, on='EmpID', how='outer'))
```

```
   EmpID   Name     Dept
0      1    Raj      NaN
1      2   Amit       HR
2      3  Priya       IT
3      4    NaN  Finance
```

📝 Hinglish: Employee aur Department table ko join karke ek combined table ban gaya.

📌 (iii) What is Join?

📏 **Definition — Join ka matlab hai do DataFrames ko index ya key column ke base par combine karna. Ye bhi SQL join ki tarah hota hai, lekin zyada tar index-based operations ke liye use hota hai.**

📖 English:

Join is used to combine two DataFrames based on their **index (by default)** or a key column. Ye SQL-style joins jaisa hi hota hai — **left**, **right**, **outer**, ya **inner**.

👉 **df1.join(df2)** → Index ke base par join karega (default = left join).

👉 **df1.join(df2, how="outer")** → Dono DataFrames ke indexes ka full outer join karega.

🗣️ Hinglish:

Join matlab DataFrames ko unke **index** ke base par jodna. Agar chaho to column ke base pe bhi kar sakte ho, lekin default index hota hai.

👉 Example: **df1.join(df2)** → Left DataFrame ke saare rows + Right wale matching indexes.

👉 Example: **df1.join(df2, how="right")** → Right DataFrame ke saare rows + Left wale matching indexes.

⚡ Ye useful hai jab DataFrames ka index meaningful ho (jaise dates, IDs), aur hume unko sidhe index ke basis pe combine karna ho.

- 🔹 Syntax:

```
DataFrame.join(
    other=df2,
    on=None,
    how="left",
    lsuffix="",
    rsuffix=""
)
```

- **Parameters**:

👉 **other** → DataFrame to join

👉 **on** → Column name to join on (optional, mostly index hota hai)

👉 **how** → "left" (default), "right", "outer", "inner"

👉 **lsuffix / rsuffix** → Same column names ke liye suffix add karne ke liye

- **Returns** → New joined **DataFrame**

📌 Real-Life Examples of Join

🍎 1. Simple Join on Index (Default Left Join)

In [14]:

```
import pandas as pd

df1 = pd.DataFrame({"Name":["A","B","C"]}, index=[1,2,3])
df2 = pd.DataFrame({"Age":[20,25,30]}, index=[2,3,4])

print(df1.join(df2))
```

```
  Name   Age
1    A   NaN
2    B  20.0
3    C  25.0
```

📝 Hinglish: df1 ke saare rows aaye, matching index par Age add ho gaya, baki NaN.

🚗 2. Inner Join

In [16]:

```
print(df1.join(df2, how='inner'))
```

```
  Name  Age
2    B   20
3    C   25
```

📝 Hinglish: Sirf un indexes par join hua jo dono DataFrames me common the (2,3).

📊 3. Outer Join

In [17]:

```
print(df1.join(df2, how='outer'))
```

```
  Name   Age
1    A   NaN
2    B  20.0
3    C  25.0
4  NaN  30.0
```

📝 Hinglish: Dono indexes ke saare rows combine hue, missing jagah NaN aa gaya.

⚽ 4. Right Join

In [18]:

```
print(df1.join(df2, how='right'))
```

```
  Name  Age
2    B   20
3    C   25
4  NaN   30
```

📝 Hinglish: df2 ke saare rows aaye, df1 ke matching Name add hue.

💰 5. Real-Life Example: Employee Salary Analysis

In [19]:

```
emp = pd.DataFrame({"Name":["Raj","Amit","Priya"]}, index=[1,2,3])
salary = pd.DataFrame({"Salary":[50000,60000,70000]}, index=[2,3,4])

print(emp.join(salary, how='outer'))
```

```
    Name   Salary
1    Raj      NaN
2   Amit  50000.0
3  Priya  60000.0
4    NaN  70000.0
```

📝 Hinglish: Employee aur Salary table ko index ke base pe join kiya — sab combine ho gaya, missing values NaN.

9. 🎯 Advanced Indexing – Powerful Index Operations

Operation	Definition	Syntax / Example
MultiIndex	Create hierarchical index for rows or columns.	`df.set_index(["region",'`

Operation	Definition	Syntax / Example
		`df.index.names`

Cross-section (xs)	Select data at particular level of MultiIndex.	`df.xs("North", level="region")`
		`df.xs(("North","Delhi"))`

Stack	Convert columns into row-level index.	`df.stack()`
Unstack	Convert row-level index back into columns.	`df.unstack()`

Reindex	Conform DataFrame to new index with optional filling logic.	`df.reindex([0,1,2,5])`
		`df.reindex(columns=["A","B","C"])`

📌 (i) What is MultiIndex?

📏 **Definition — MultiIndex Pandas ka hierarchical index hota hai jo rows ya columns ke liye ek se zyada levels of index banata hai. Ye complex datasets me grouping aur slicing ko easy banata hai.**

📖 English:

MultiIndex allows you to use **multiple levels of indexing** for rows and/or columns. Matlab ek DataFrame me nested structure jaisa index create ho jata hai.

👉 **df.set_index(["Region","City"])** → Rows ka index Region aur City dono ban jaate hain.

👉 **df.index.names** → MultiIndex ke level names check karne ke liye.

🗣️ Hinglish:

MultiIndex ek **hierarchical index** hai jisme tum ek se zyada columns ko index bana sakte ho. Jaise pehle Region, uske andar City. Isse complex data ko handle karna easy ho jata hai.

👉 Example: **df.set_index(["Dept","Team"])** → Dept aur Team dono ko ek saath index bana diya.

👉 Example: **df.index.names** → Dono index levels ke naam dekhne ke liye.

⚡ Ye useful hai jab tumhe multi-level grouping karni ho ya pivot table jaisa structure maintain karna ho.

🔹 Syntax:

```
DataFrame.set_index(
    keys=["col1","col2"],
    inplace=False
)

DataFrame.index.names
```

- **Parameters:**

👉 **keys** → Column(s) jo index banenge

👉 **inplace** → True karne pe DataFrame modify ho jayega

👉 **names** → MultiIndex levels ke naam

- **Returns** → DataFrame with **MultiIndex**

📌 Real-Life Examples of MultiIndex

🍎 1. Simple MultiIndex by Two Columns

In [1]:

```
import pandas as pd

df = pd.DataFrame({
    "Region":["East","East","West","West"],
    "City":["Delhi","Kolkata","Mumbai","Pune"],
    "Sales":[100,200,300,400]
})

df2 = df.set_index(['Region', 'City'])
print(df2)
```

```
                Sales
Region City
East   Delhi      100
       Kolkata    200
West   Mumbai     300
       Pune       400
```

📝 Hinglish: Region aur City dono ko index bana diya, ab hierarchical index hai.

🚗 2. Access Data from MultiIndex

In [5]:

```
print(df2.loc['East'])
print("\n", df2.loc[('West', 'Mumbai')])
```

```
         Sales
City
Delhi      100
Kolkata    200

 Sales    300
Name: (West, Mumbai), dtype: int64
```

📝 Hinglish: Pehle East region ka pura data nikal diya, fir West → Mumbai ka specific row.

📊 3. MultiIndex Columns

In [8]:

```
arrays = [["Maths","Maths","Science","Science"],
          ["Midterm","Final","Midterm","Final"]]

index = pd.MultiIndex.from_arrays(arrays, names=('Subject', 'Exam'))

df = pd.DataFrame([[80,90,85,95],[70,75,65,80]],
                  index=["Alice","Bob"],
                  columns=index)

print(df)
```

```
Subject   Maths         Science
Exam    Midterm Final Midterm Final
Alice        80    90      85    95
Bob          70    75      65    80
```

📝 Hinglish: Columns me MultiIndex ban gaya → Subject aur Exam dono levels hai.

⚽ 4. Reset MultiIndex

In [9]:

```
print(df2.reset_index())
```

```
  Region     City  Sales
0   East    Delhi    100
1   East  Kolkata    200
2   West   Mumbai    300
3   West     Pune    400
```

📝 Hinglish: MultiIndex hata ke columns wapas normal ban gaye.

💰 5. Real-Life Example: Department & Team Data

In [11]:

```
df = pd.DataFrame({
    "Dept":["IT","IT","HR","HR"],
    "Team":["A","B","A","B"],
    "Salary":[50000,60000,40000,45000]
})
```

```
df2 = df.set_index(['Dept', 'Team'])
print(df2)
```

```
           Salary
Dept Team        
IT   A      50000
     B      60000
HR   A      40000
     B      45000
```

📝 Hinglish: Department aur Team ko hierarchical index bana diya salary ke analysis ke liye.

📌 (ii) What is Cross-section (xs)?

📏 **Definition — xs() ka use tab hota hai jab tumhe MultiIndex wale DataFrame se kisi ek particular level ka slice nikalna ho. Matlab tum directly bol sakte ho → bhai mujhe sirf "North" region ka data do ya mujhe ("North","Delhi") ka ek hi row chahiye.**

📖 English:

The .xs() method allows you to quickly select rows (or columns) at a specific level of a **MultiIndex**.
Instead of writing long .loc[] combinations, you can extract precise data in just one line.

👉 **df.xs("North", level="Region")** → Sirf "North" region ka data dega.
👉 **df.xs(("North","Delhi"), level=[0,1])** → Region=North aur City=Delhi ka row return karega.

🗣️ Hinglish:

xs() ek shortcut hai jo directly MultiIndex ke andar ghus ke data nikal leta hai. Jaise ek bada cupboard ho jisme shelves aur boxes ho, aur tum seedha bol do → "shelf = North, box = Delhi", aur turant wahi data mil jaye.

👉 Example: **df.xs("East", level="Region")** → East region ka saara data.
👉 Example: **df.xs("Mumbai", level="City")** → Sirf Mumbai ka data.

⚡ Ye useful hai jab MultiIndex ke andar deep jaake specific slice chahiye bina long code likhe.

🔹 Syntax:

```
DataFrame.xs(
    key="North",
    axis=0,
    level="Region",
    drop_level=True
)
```

- **Parameters**:

👉 **key** → Value jo select karni hai (e.g., "North")

👉 **axis** → 0 (rows) ya 1 (columns), default = rows

👉 **level** → MultiIndex ka konsa level target karna hai (e.g., "Region")

👉 **drop_level** → True matlab selected level result se remove hoga, False matlab preserve hoga

- **Returns** → Extracted slice from **DataFrame**

📌 Real-Life Examples of xs()

🍎 1. Basic MultiIndex Selection (Single Level)

In [12]:

```
import pandas as pd

df = pd.DataFrame({
    "Region":["North","North","South","South"],
    "City":["Delhi","Lucknow","Chennai","Bangalore"],
    "Sales":[250,300,200,400]
})

df2 = df.set_index(['Region', 'City'])
print(df2)

# Cross-section by region
print(df2.xs('North'))
```

```
                  Sales
Region City
North  Delhi        250
       Lucknow      300
South  Chennai      200
       Bangalore    400
         Sales
City
Delhi      250
Lucknow    300
```

📝 Hinglish: Yaha pe `xs("North")` ne sirf North region ka pura data de diya.

🚗 2. Multi-Level Selection

In [13]:

```
# Single row select with tuple
print(df2.xs(('North', 'Delhi')))
```

```
Sales    250
Name: (North, Delhi), dtype: int64
```

📝 Hinglish: Yaha pe direct North → Delhi ka ek hi row nikal liya.

📊 3. Selection by Level Name

In [14]:

```
print(df2.xs('Delhi', level='City'))
```

```
        Sales
Region
North     250
```

📝 Hinglish: Ab sirf **City = Delhi** wala data nikal aaya, irrespective of region.

⚽ 4. Using drop_level=False

In [16]:

```
print(df2.xs('North', level='Region', drop_level=False))
```

```
                Sales
Region City
North  Delhi      250
       Lucknow    300
```

📝 Hinglish: Agar tum chahte ho ki result me bhi "Region" ka level preserve ho, to `drop_level=False` use karo.

💰 5. Cross-section in Columns (axis=1)

In [20]:

```
arrays = [["Maths","Maths","Science","Science"],
          ["Midterm","Final","Midterm","Final"]]

cols = pd.MultiIndex.from_arrays(arrays, names=("Subject","Exam"))

df = pd.DataFrame([[80,90,85,95],[70,75,65,80]],
                  index=["Alice","Bob"],
                  columns=cols)

print(df)

# Select all "Maths" marks
print('\n',df.xs("Maths", axis=1, level="Subject"))
```

```
Subject   Maths         Science
Exam    Midterm Final Midterm Final
Alice        80    90      85    95
Bob          70    75      65    80

 Exam   Midterm  Final
Alice       80     90
Bob         70     75
```

📝 Hinglish: Isme MultiIndex columns the → xs ne Maths subject ka pura data de diya.

📌 (iii) What is stack()?

📏 **Definition — stack() ka use hota hai columns ko row-level index me convert karne ke liye. Matlab columns ko ek additional index level bana diya jata hai, aur DataFrame long-format me aa jata hai.**

📖 English:

The **.stack()** method pivots the columns of a DataFrame into the index, producing a Series (or DataFrame if multiple levels).
It's super useful for reshaping **wide-format** data into **long-format**.

👉 **df.stack()** → Sabhi columns ko rows ke andar stack kar dega.
👉 **df.stack(level=1)** → Sirf ek specific level of columns ko stack karega.

🗣️ Hinglish:

Socho tumhare paas ek table hai jisme subjects alag-alag columns me hain (Maths, Science), aur tum chahte ho ki ye subjects rows me aa jaye.
Tab **stack()** bolta hai: *"Bhai columns side me mat rakho, niche rows ke saath stack kardo."*

👉 Example: **df.stack()** → Maths aur Science dono row-index me aa jayenge.
👉 Example: **df.stack(dropna=False)** → NaN values bhi stack ho jaayengi.

⚡ Ye reshaping ke liye sabse simple aur powerful method hai jab tumhe data analysis ya visualization ke liye long-format chahiye.

🔹 Syntax:

```
DataFrame.stack(
    level=-1,
    dropna=True
)
```

- **Parameters:**

👉 **level** → Kis column level ko stack karna hai (default = last)

👉 **dropna** → True par NaN remove honge, False par NaN bhi stack honge

- **Returns** → Long-format **DataFrame/Series**

📌 Real-Life Examples of stack()

🍎 1. Basic Stack Example

In [22]:

```
import pandas as pd

df = pd.DataFrame({
    "Name":["Alice","Bob"],
    "Maths":[80,70],
    "Science":[90,65]
})

print("Original:\n", df)

# Apply stack
print("Stacked:\n\n", df.set_index("Name").stack())
```

```
Original:
     Name  Maths  Science
0  Alice     80       90
1    Bob     70       65
Stacked:

 Name
Alice  Maths      80
       Science    90
Bob    Maths      70
       Science    65
dtype: int64
```

📝 Hinglish: Yaha Maths aur Science columns stack ho gaye aur ek hierarchical index ban gaya → Name + Subject.

🚗 2. MultiIndex Columns with Stack

In [24]:

```
arrays = [["Maths","Maths","Science","Science"],
          ["Midterm","Final","Midterm","Final"]]

cols = pd.MultiIndex.from_arrays(arrays, names=("Subject","Exam"))

df = pd.DataFrame([[80,90,85,95],[70,75,65,80]],
                  index=["Alice","Bob"],
                  columns=cols)

print("Original:\n", df)

# Stack one level
print("Stacked on Exam:\n\n", df.stack(level="Exam"))
```

```
Original:
 Subject   Maths        Science
Exam     Midterm Final Midterm Final
Alice         80    90      85    95
Bob           70    75      65    80
Stacked on Exam:

 Subject        Maths  Science
      Exam
Alice Final        90       95
      Midterm      80       85
Bob   Final        75       80
      Midterm      70       65
C:\Users\gkuir\AppData\Local\Temp\ipykernel_8236\3292008110.py:13: FutureWarning: The pr
evious implementation of stack is deprecated and will be removed in a future version of
pandas. See the What's New notes for pandas 2.1.0 for details. Specify future_stack=True
to adopt the new implementation and silence this warning.
  print("Stacked on Exam:\n\n", df.stack(level="Exam"))
```

📝 Hinglish: Yaha Exam level ko stack kar diya → ab Midterm aur Final row index ban gaye.

📊 3. Stack with dropna=False

In [28]:

```python
df = pd.DataFrame({
    "A":[1,None],
    "B":[3,4]
}, index=["x","y"])

print(df.stack(dropna=False))
```

```
x  A    1.0
   B    3.0
y  A    NaN
   B    4.0
dtype: float64
C:\Users\gkuir\AppData\Local\Temp\ipykernel_8236\3572751257.py:6: FutureWarning: The pre
vious implementation of stack is deprecated and will be removed in a future version of p
andas. See the What's New notes for pandas 2.1.0 for details. Specify future_stack=True
to adopt the new implementation and silence this warning.
  print(df.stack(dropna=False))
```

📝 Hinglish: Agar `dropna=False` karte ho to NaN bhi stack ho jaata hai.

💰 5. Real-Life Example: Sales Data

In [35]:

```python
df = pd.DataFrame({
    "Region":["North","South"],
    "Q1":[200,150],
    "Q2":[250,180],
    "Q3":[300,200]
})

df2 = df.set_index("Region")
print("Stacked Sales:\n", df2.stack())
```

```
Stacked Sales:
 Region
```

```
North   Q1    200
        Q2    250
        Q3    300
South   Q1    150
        Q2    180
        Q3    200
dtype: int64
```

📝 Hinglish: Yaha quarterly sales jo alag columns me the, wo rows ke andar aa gaye analysis ke liye.

📌 (iv) What is unstack()?

📏 **Definition — unstack() ka use hota hai row-level index ko wapas columns me convert karne ke liye. Matlab jo tumne stack() karke long-format banaya tha, usko unstack() karke fir se wide-format bana sakte ho.**

📖 English:

The **.unstack()** method pivots a level of the row index into columns, reshaping the DataFrame back into **wide format**.
It's basically the reverse of **stack()**.

👉 **df.unstack()** → Last row-level index ko columns me convert karega.
👉 **df.unstack(level=0)** → Specific index level ko columns me pivot karega.

🗣️ Hinglish:

Socho tumne **Maths** aur **Science** ko **stack()** karke rows me bhej diya tha. Ab **unstack()** bolta hai → *"Bhai chinta mat kar, mai unhe wapas columns ki kursi pe bitha deta hoon."*

👉 Example: **df.stack().unstack()** → Original wide-format wapas aa jayega.
👉 Example: **df.unstack(fill_value=0)** → Missing values ko 0 se fill karega.

⚡ Ye reshaping ke liye useful hai jab tumhe grouped data ko wapas columnar (wide) format me laana ho.

🔹 Syntax:

```
DataFrame.unstack(
    level=-1,
    fill_value=None
)
```

- **Parameters**:

👉 **level** → Index ka konsa level columns banega (default = last)

👉 **fill_value** → Missing values ko fill karne ke liye default value

- **Returns** → Wide-format **DataFrame**

📌 Real-Life Examples of unstack()

🍎 1. Basic Unstack Example

In [38]:

```
import pandas as pd

df = pd.DataFrame({
    "Name":["Alice","Bob"],
    "Maths":[80,70],
    "Science":[90,65]
})

stacked = df.set_index("Name").stack()
print("Stacked:\n", stacked)

# Apply unstack
print("Unstacked:\n\n", stacked.unstack())
```

```
Stacked:
 Name
Alice  Maths      80
       Science    90
Bob    Maths      70
       Science    65
dtype: int64
Unstacked:

        Maths  Science
Name
Alice     80       90
Bob       70       65
```

📝 Hinglish: Pehle stack karke Maths & Science ko rows me bheja, fir unstack karke wapas columns me le aaye.

🚗 2. MultiIndex with Unstack

In [40]:

```
arrays = [["Maths","Maths","Science","Science"],
          ["Midterm","Final","Midterm","Final"]]
```

```
cols = pd.MultiIndex.from_arrays(arrays, names=("Subject","Exam"))

df = pd.DataFrame([[80,90,85,95],[70,75,65,80]],
                  index=["Alice","Bob"],
                  columns=cols)

# Stack Exam level
stacked = df.stack(level="Exam")
print("Stacked:\n", stacked)

# Unstack Exam level
print("Unstacked:\n\n", stacked.unstack(level="Exam"))
```

```
Stacked:
 Subject       Maths  Science
      Exam
Alice Final       90       95
      Midterm     80       85
Bob   Final       75       80
      Midterm     70       65
Unstacked:

 Subject Maths         Science
Exam     Final Midterm   Final Midterm
Alice       90      80      95      85
Bob         75      70      80      65
C:\Users\gkuir\AppData\Local\Temp\ipykernel_8236\1248799967.py:11: FutureWarning: The pr
evious implementation of stack is deprecated and will be removed in a future version of
pandas. See the What's New notes for pandas 2.1.0 for details. Specify future_stack=True
to adopt the new implementation and silence this warning.
  stacked = df.stack(level="Exam")
```

📝 Hinglish: Exam ko stack kiya to Midterm/Final rows ban gaye. Fir unstack karke wapas columns bana diye.

📊 3. Unstack Specific Level

In [42]:

```
df = pd.DataFrame({
    "Region":["North","North","South","South"],
    "City":["Delhi","Lucknow","Chennai","Bangalore"],
    "Sales":[250,300,200,400]
})

df2 = df.set_index(["Region","City"])
stacked = df2.stack()
print(stacked)
print("Unstack City Level:\n\n", stacked.unstack(level="City"))
```

```
Region  City
North   Delhi      Sales    250
        Lucknow    Sales    300
South   Chennai    Sales    200
        Bangalore  Sales    400
dtype: int64
Unstack City Level:

 City         Bangalore  Chennai  Delhi  Lucknow
Region
```

```
North  Sales        NaN      NaN  250.0    300.0
South  Sales      400.0    200.0    NaN      NaN
```

📝 Hinglish: Yaha City level ko unstack karke har ek City column ban gaya.

⚽ 4. Unstack with fill_value

In [43]:

```
df = pd.DataFrame({
    "Region":["North","South"],
    "Q1":[200,150],
    "Q2":[250,None]
})

stacked = df.set_index("Region").stack()
print("Unstacked with fill:\n", stacked.unstack(fill_value=0))
```

```
Unstacked with fill:
            Q1     Q2
Region
North   200.0  250.0
South   150.0    0.0
```

📝 Hinglish: Missing Q2 ko `fill_value=0` se bhar diya.

💰 5. Real-Life Example: Employee Department

In [44]:

```
df = pd.DataFrame({
    "Dept":["IT","IT","HR","HR"],
    "Team":["A","B","A","B"],
    "Salary":[50000,60000,40000,45000]
})

df2 = df.set_index(["Dept","Team"]).stack()
print("Stacked Salaries:\n", df2)

print("Unstacked Salaries:\n", df2.unstack())
```

```
Stacked Salaries:
 Dept  Team
IT    A     Salary    50000
      B     Salary    60000
HR    A     Salary    40000
      B     Salary    45000
dtype: int64
Unstacked Salaries:
            Salary
Dept Team
HR   A      40000
     B      45000
IT   A      50000
     B      60000
```

📝 Hinglish: Salary data ko stack karke row-level pe bheja, fir unstack karke wapas wide-format me laa diya analysis ke liye.

📌 (v) What is reindex()?

📏 **Definition — reindex() ka use hota hai DataFrame ya Series ko naye index ke according align karne ke liye. Matlab agar tumhare paas missing index values hain ya naye order me arrange karna hai, to reindex() bolta hai → *"Bhai, bata kaunsa order chahiye, mai adjust kar deta hoon."***

📖 English:

The **.reindex()** method conforms a DataFrame to a new index or new set of columns.
It can insert missing values (**NaN**) where data is not available and can also fill values using different strategies.

👉 **df.reindex([0,1,2,5])** → Agar 3 missing hai, to NaN aa jayega.
👉 **df.reindex(columns=["A","B","C"])** → Data ko naye column labels ke hisaab se adjust karega.

🗣️ Hinglish:

Socho tumhare paas roll numbers 0,1,2,3 ke data hai, lekin tumhe 0,1,2,5 chahiye.
To **reindex()** bolega → *"3 missing hai, mai uski jagah NaN rakh deta hoon."*

👉 Example: **df.reindex([0,1,2,5], fill_value=0)** → NaN ki jagah 0 daal dega.
👉 Example: **df.reindex(index=[0,1,2], method="ffill")** → Forward fill karega.

⚡ Ye tab kaam aata hai jab tumhe index reorder karna ho ya missing labels ke liye default values dalni ho.

🔹 Syntax:

```
DataFrame.reindex(
    labels=None,
    index=None,
    columns=None,
    axis=None,
    method=None,
    fill_value=None
)
```

- **Parameters:**

  👉 **index / columns** → New index ya new column labels specify karne ke liye

  👉 **axis** → 0 = rows, 1 = columns

  👉 **method** → Missing values fill karne ka tarika: **ffill** (forward fill), **bfill** (backward fill)

  👉 **fill_value** → NaN ke jagah koi default value

- **Returns** → Reindexed **DataFrame/Series**

📌 Real-Life Examples of reindex()

🍎 1. Basic Row Reindex

In [45]:

```
import pandas as pd

df = pd.DataFrame({"A":[10,20,30]}, index=[0,1,2])
print("Original:\n", df)

# Reindex with a new index
print("Reindexed:\n", df.reindex([0, 1, 2, 5]))
```

```
Original:
     A
0   10
1   20
2   30
Reindexed:
         A
0   10.0
1   20.0
2   30.0
5    NaN
```

📝 Hinglish: Index 5 pe koi data nahi tha → NaN aa gaya.

🚗 2. Reindex Columns

In [47]:

```
df = pd.DataFrame({"A":[1,2], "B":[3,4]})
print("Reindexed Columns:\n", df.reindex(columns=['A', 'B', 'C']))
```

```
Reindexed Columns:
    A  B   C
0  1  3 NaN
1  2  4 NaN
```

📝 Hinglish: Column `C` missing tha, isliye NaN se bhar diya.

📊 3. Reindex with fill_value

In [50]:

```
df = pd.DataFrame({"A":[10,20,30]}, index=[0,1,2])
print(df)

print('\n',df.reindex([0,1,2,5], fill_value=0))
```

```
    A
0  10
1  20
2  30

     A
0  10
1  20
2  30
5   0
```

📝 Hinglish: Missing row (index=5) ko NaN ki jagah `0` se bhar diya.

⚽ 4. Reindex with ffill (forward fill)

In [52]:

```
df = pd.DataFrame({'A':[100, 200, 300]}, index=[0,1,2])
print(df)

print('\n',df.reindex([0,1,2,4,6], method='ffill') )
```

```
     A
0  100
1  200
2  300

      A
0  100
1  200
2  300
4  300
6  300
```

📝 Hinglish: Extra rows 3 aur 4 ke liye pichle value ko aage copy kar diya.

💡 5. Reindex with bfill (backward fill)

In [54]:

```
df = pd.DataFrame({'A':[100, 300, 456, 789]}, index=[0,1,3,4])
print(df.reindex([1,2,3,4,5], method='bfill'))
```

```
       A
1  300.0
2  456.0
3  456.0
4  789.0
5    NaN
```

📝 Hinglish: Extra rows me aage ke value ko peeche copy karke fill kar diya.

💰 6. Real-Life Example: Stock Prices

In [56]:

```
df = pd.DataFrame({
    "Price":[100,105,110],
    "Date":["2023-01-01","2023-01-02","2023-01-04"]
}).set_index("Date")

new_index = ["2023-01-01","2023-01-02","2023-01-03","2023-01-04"]

print(df)

print(df.reindex(new_index, method='ffill'))
```

```
            Price
Date
2023-01-01    100
2023-01-02    105
2023-01-04    110
            Price
Date
2023-01-01    100
2023-01-02    105
2023-01-03    105
2023-01-04    110
```

📝 Hinglish: 3rd Jan ka data missing tha → forward fill use karke 2nd Jan ka price copy kar diya.

10. ⏳ Time Series Handling – Work with Dates & Times

Operation	Definition	Syntax / Example

DateTime Conversion	Convert strings or numbers to datetime format.	`pd.to_datetime(df["date'`
		`pd.date_range("2023-01-01", periods=5, freq="D")`

Date-based Indexing	Index and slice data using datetime.	`df.loc["2023-01-01"]`
		`df["2023-01":"2023-03"]`

Resampling	Change frequency of time series data (upsample/downsample).	`df.resample("M").sum()`
		`df.resample("W").mean()`

Rolling Windows	Perform moving average or rolling calculations.	`df["sales"].rolling(7).m`
		`df["temp"].rolling(windo`

📌 (i) What is DateTime Conversion?

📏 **Definition — DateTime Conversion ka use hota hai jab tumhe string ya numeric values ko proper date-time object me badalna ho.**
Pandas pd.to_datetime() aur pd.date_range() jaise functions deta hai jo time-series analysis ke liye powerful tools hai.

📖 English:

The **.to_datetime()** function converts strings, numbers, or mixed formats into **datetime64** objects, which Pandas understands for time-series operations.
Similarly, **pd.date_range()** generates a fixed-frequency sequence of dates.

👉 **pd.to_datetime(["2023-01-01","2023-01-02"])** → Strings ko datetime objects me convert karega.
👉 **pd.date_range(start="2023-01-01", periods=5, freq="D")** → Daily frequency ka range generate karega.

🗣️ Hinglish:

Socho tumhare paas **"2023-01-01"** string likha hai, lekin Pandas ko tab tak samajh nahi aata jab tak usko proper date format me convert na karo.
Tab **pd.to_datetime()** bolta hai → *"Bhai, tension mat le, main is string ko asli date bana deta hoon."*

👉 Example: **pd.to_datetime("2023-03-15")** → Ek single date ko datetime object me badal dega.
👉 Example: **pd.date_range("2023-01-01", "2023-01-07")** → 7 din ka continuous range bana dega.

🔹 Syntax:

```
pd.to_datetime(
    arg,
    format=None,
    errors="raise"
)
```

```
pd.date_range(
start=None,
end=None,
periods=None,
freq=None
)
```

- **Parameters**:

👉 **arg** → String, list, Series ya array jisme date-like values ho

👉 **format** → Date ka format specify karne ke liye (jaise "%Y-%m-%d")

👉 **errors** → "raise" (error throw kare), "coerce" (invalid ko NaT banaye)

👉 **start / end** → Range ke start aur end dates

👉 **periods** → Kitni dates generate karni hain

👉 **freq** → Frequency (D = Day, M = Month, H = Hour, etc.)

- **Returns** → **DatetimeIndex** ya **Series of datetime64**

📌 Real-Life Examples of DateTime Conversion

🍎 1. Basic String to DateTime

In [1]:

```
import pandas as pd
dates = ['2023-01-01', '2023-01-02', '2023-01-03']
print(pd.to_datetime(dates))
```

```
DatetimeIndex(['2023-01-01', '2023-01-02', '2023-01-03'], dtype='datetime64[ns]', freq=N
one)
```

📝 Hinglish: Strings ko directly datetime object me convert kar diya.

🚗 2. Custom Format Conversion

In [4]:

```
dates = ['01-02-2023', '05-02-2023']
print(pd.to_datetime(dates, format="%d-%m-%Y"))
```

```
DatetimeIndex(['2023-02-01', '2023-02-05'], dtype='datetime64[ns]', freq=None)
```

📝 Hinglish: Yaha format manually diya ("%d-%m-%Y") → din-pehle, fir month, fir year.

📊 3. Handle Invalid Dates (errors="coerce")

```
In [6]:
```

```python
dates = ["2023-01-01", "not_a_date", "2023-01-03"]
print(pd.to_datetime(dates, errors='coerce'))
```

```
DatetimeIndex(['2023-01-01', 'NaT', '2023-01-03'], dtype='datetime64[ns]', freq=None)
```

📝 Hinglish: Invalid value "not_a_date" ko NaT (Not a Time) bana diya.

⚽ 4. Generate Date Range

```
In [9]:
```

```python
print(pd.date_range("2023-01-01", periods=5, freq="D"))
```

```
DatetimeIndex(['2023-01-01', '2023-01-02', '2023-01-03', '2023-01-04',
               '2023-01-05'],
              dtype='datetime64[ns]', freq='D')
```

📝 Hinglish: 1st Jan se start karke 5 continuous din ka range bana diya.

💡 5. Different Frequencies in Date Range

```
In [11]:
```

```python
print(pd.date_range('2023-01-01', periods=4, freq="M"))
print(pd.date_range('2023-01-01', periods=4, freq="H"))
```

```
DatetimeIndex(['2023-01-31', '2023-02-28', '2023-03-31', '2023-04-30'], dtype='datetime6
4[ns]', freq='ME')
DatetimeIndex(['2023-01-01 00:00:00', '2023-01-01 01:00:00',
               '2023-01-01 02:00:00', '2023-01-01 03:00:00'],
              dtype='datetime64[ns]', freq='h')
C:\Users\gkuir\AppData\Local\Temp\ipykernel_15132\838310684.py:1: FutureWarning: 'M' is
deprecated and will be removed in a future version, please use 'ME' instead.
  print(pd.date_range('2023-01-01', periods=4, freq="M"))
C:\Users\gkuir\AppData\Local\Temp\ipykernel_15132\838310684.py:2: FutureWarning: 'H' is
deprecated and will be removed in a future version, please use 'h' instead.
  print(pd.date_range('2023-01-01', periods=4, freq="H"))
```

📝 Hinglish: Pehle monthly dates banaye, fir hourly dates. Frequency change kar ke tum apni need ke hisaab se bana sakte ho.

💰 6. Real-Life Example: Stock Data Dates

```
In [18]:
```

```python
df = pd.DataFrame({
    "Price":[100, 105, 110],
    'Date':['2023-01-01', '2023-01-02', '2023-01-04']
})

print(df.dtypes)
df['Date']=pd.to_datetime(df['Date'])
print('\n',df)
print('\n', df.dtypes)
```

```
Price     int64
Date     object
dtype: object

    Price       Date
```

```
0    100 2023-01-01
1    105 2023-01-02
2    110 2023-01-04

 Price            int64
Date     datetime64[ns]
dtype: object
```

📝 Hinglish: Stock prices ki Date column ko string se datetime object me convert kiya, ab tum time-series analysis kar sakte ho.

📌 (ii) What is Date-based Indexing?

📏 **Definition — Date-based Indexing ka use hota hai jab tum DataFrame ya Series ko datetime index ke basis par slice ya filter karna chahte ho.**
Matlab agar tumhare paas daily stock prices hain aur tumhe sirf "2023-01-01" ka data chahiye ya "2023-01" se "2023-03" tak ka data chahiye → to .loc[] aur slicing ke saath date-based indexing use hoti hai.

📖 English:

Date-based indexing allows you to select and slice data using **date strings, partial dates (year/month), or full date ranges**.
Works only when the DataFrame has a **DateTimeIndex**.

👉 **df.loc["2023-01-01"]** → Select data of single date.
👉 **df["2023-01":"2023-03"]** → Select data from Jan to Mar 2023.
👉 **df["2023-01"]** → Select all data of January 2023.
👉 **df["2023"]** → Select all data of year 2023.

🗣️ Hinglish:

Socho tumhare paas ek **calendar** hai. Tum keh sakte ho:

👉 *"Bhai mujhe 1st Jan ka page dikha do."*
👉 *"Ya mujhe Jan se March tak ka pura data chahiye."*

Ye hi kaam **Pandas date-based indexing** karta hai.

- 🔹 Syntax:

```
Single date selection
df.loc["2023-01-01"]


Date range selection
df["2023-01":"2023-03"]


Partial string indexing
df["2023-01"]   # January 2023 ka pura data
df["2023"]      # 2023 ka pura data
```

- **Parameters / Notes**:

👉 Works only if DataFrame ka index **DateTimeIndex** ho.

👉 Partial string matching supported → **"2023"** = full year, **"2023-01"** = full month.

- **Returns** → Sliced **DataFrame/Series** based on date range

📌 Real-Life Examples of Date-based Indexing

🍎 1. Single Date Selection

In [22]:

```
import pandas as pd
dates = pd.date_range("2023-01-01", periods=5, freq="D")
df = pd.DataFrame({'Sales':[100, 200, 300, 400, 500]}, index=dates)

print(df)
print('\n', df.loc["2023-01-01"])
```

```
            Sales
2023-01-01    100
2023-01-02    200
2023-01-03    300
2023-01-04    400
2023-01-05    500

 Sales    100
Name: 2023-01-01 00:00:00, dtype: int64
```

📝 Hinglish: Sirf 1st Jan ka sales data uthaya.

🚗 2. Date Range Selection

In [23]:

```
print(df['2023-01-01':'2023-01-03'])
```

```
            Sales
2023-01-01    100
2023-01-02    200
2023-01-03    300
```

📝 Hinglish: 1st Jan se 3rd Jan tak ka data mil gaya, dono ends inclusive hote hain.

📊 3. Month-wise Selection (Partial String)

In [29]:

```
print(df.loc["2023-01"])
```

```
            Sales
Date             
2023-01-01      0
2023-01-02      1
2023-01-03      2
2023-01-04      3
2023-01-05      4
2023-01-06      5
2023-01-07      6
2023-01-08      7
2023-01-09      8
2023-01-10      9
2023-01-11     10
2023-01-12     11
2023-01-13     12
2023-01-14     13
2023-01-15     14
2023-01-16     15
2023-01-17     16
2023-01-18     17
2023-01-19     18
2023-01-20     19
2023-01-21     20
2023-01-22     21
2023-01-23     22
2023-01-24     23
2023-01-25     24
2023-01-26     25
2023-01-27     26
2023-01-28     27
2023-01-29     28
2023-01-30     29
2023-01-31     30
```

📝 Hinglish: `"2023-01"` likhne se pura January ka data aa gaya.

⚽ 4. Year-wise Selection

In [31]:

```
dates = pd.date_range("2022-12-25", periods=400, freq="D")
df = pd.DataFrame({"Visitors":range(400)}, index=dates)

print(df.loc["2023"])
```

```
            Visitors
2023-01-01         7
2023-01-02         8
```

```
2023-01-03         9
2023-01-04        10
2023-01-05        11
...              ...
2023-12-27       367
2023-12-28       368
2023-12-29       369
2023-12-30       370
2023-12-31       371

[365 rows x 1 columns]
```

📝 Hinglish: Sirf 2023 ke saal ka data select ho gaya.

💡 5. Between Specific Dates

In [32]:

```
print(df.loc['2023-03-01':'2023-06-30'])
```

```
            Visitors
2023-03-01        66
2023-03-02        67
2023-03-03        68
2023-03-04        69
2023-03-05        70
...              ...
2023-06-26       183
2023-06-27       184
2023-06-28       185
2023-06-29       186
2023-06-30       187

[122 rows x 1 columns]
```

📝 Hinglish: March se June 2023 tak ka data extract kar liya.

💰 6. Real-Life Example: Stock Prices Filtering

In [33]:

```
df = pd.DataFrame({
    "Price":[100,105,110,120,125],
}, index=pd.date_range("2023-01-01", periods=5, freq="D"))

# Select January data
print(df.loc['2023-01'])
```

```
            Price
2023-01-01    100
2023-01-02    105
2023-01-03    110
2023-01-04    120
2023-01-05    125
```

📝 Hinglish: Stock data me sirf January ka subset le liya for analysis.

📌 (iii) What is Resampling?

📏 **Definition — Resampling ka matlab hota hai time-series data ki frequency badalna.**
Matlab tumhare paas daily data hai aur usse monthly summary chahiye (downsampling), ya fir tumhare paas yearly data hai aur usko daily/monthly expand karna hai (upsampling).

📖 English:

Resampling changes the frequency of time-series data.
👉 **Downsampling**: High frequency → Low frequency (e.g., daily → monthly).
👉 **Upsampling**: Low frequency → High frequency (e.g., monthly → daily).

👉 **df.resample("M").sum()** → Daily to monthly totals.
👉 **df.resample("D").ffill()** → Monthly to daily forward-fill.
👉 **df.resample("W").mean()** → Weekly average.

🗣️ Hinglish:

Socho tumhare paas ek **rozana ka data diary** hai.

👉 Agar tum bas **monthly total sales** dekhna chaho to downsample karoge.
👉 Agar tumko **daily details** banana hai monthly data se (chhote parts me todna hai), to upsample karoge.

Yahi kaam **resampling** karta hai.

🔹 Syntax:

```
Downsampling (daily → monthly)
df.resample("M").sum()


Upsampling (monthly → daily)
df.resample("D").ffill()


Weekly average
df.resample("W").mean()
```

- **Parameters / Notes:**

  👉 **"M"** → Month-end frequency

  👉 **"W"** → Weekly frequency

  👉 **"D"** → Daily frequency

  👉 **.sum()**, **.mean()**, **.ffill()**, **.bfill()** → Aggregation / fill methods

- **Returns** → Resampled **DataFrame/Series**

📌 Real-Life Examples of Resampling

🍎 1. Downsampling: Daily → Monthly Sales

In [34]:

```
import pandas as pd

dates = pd.date_range('2023-01-01', periods=90, freq="D")
df = pd.DataFrame({'Sales': range(1, 91)}, index=dates)

print(df.resample("M").sum())
```

```
            Sales
2023-01-31    496
2023-02-28   1274
2023-03-31   2325
C:\Users\gkuir\AppData\Local\Temp\ipykernel_15132\2778674680.py:6: FutureWarning: 'M' is
deprecated and will be removed in a future version, please use 'ME' instead.
  print(df.resample("M").sum())
```

📝 Hinglish: Roz ke sales ko jod kar **monthly total** nikal liya.

🚗 2. Weekly Average Visitors

In [35]:

```
dates = pd.date_range("2023-01-01", periods=30, freq="D")
df = pd.DataFrame({"Visitors": range(10,40)}, index=dates)

print(df.resample("W").mean())
```

```
            Visitors
2023-01-01      10.0
2023-01-08      14.0
2023-01-15      21.0
2023-01-22      28.0
2023-01-29      35.0
2023-02-05      39.0
```

📝 Hinglish: Daily visitors ko **har week ka average** banaya.

📊 3. Upsampling: Monthly → Daily

In [38]:

```
dates = pd.date_range("2023-01-01", periods=3, freq="M")
df = pd.DataFrame({"Profit":[1000,1500,2000]}, index=dates)

print(df.resample("D").ffill().head())
```

```
            Profit
2023-01-31    1000
2023-02-01    1000
2023-02-02    1000
2023-02-03    1000
2023-02-04    1000
C:\Users\gkuir\AppData\Local\Temp\ipykernel_15132\2075549325.py:1: FutureWarning: 'M' is
deprecated and will be removed in a future version, please use 'ME' instead.
  dates = pd.date_range("2023-01-01", periods=3, freq="M")
```

📝 Hinglish: Monthly profit ko har din me replicate kar diya using **forward fill (ffill).**

⚽ 4. Using Backward Fill (bfill)

In [40]:

```
print(df.resample("D").bfill().head())
```

```
            Profit
2023-01-31    1000
2023-02-01    1500
2023-02-02    1500
2023-02-03    1500
2023-02-04    1500
```

📝 Hinglish: Har din ka value agli month se **backward fill** kar diya.

💰 5. Real-Life Example: Stock Prices

In [41]:

```
df = pd.DataFrame({
    "Price":[100,105,110,120,125],
}, index=pd.date_range("2023-01-01", periods=5, freq="D"))

# Convert daily prices to weekly mean
print(df.resample("W").mean())
```

```
            Price
2023-01-01  100.0
2023-01-08  115.0
```

📝 Hinglish: Roz ke stock prices ko **weekly average price** me convert kar liya.

📌 (iv) What is Rolling Window?

📏 Definition — Rolling Window ka matlab hai ek fixed-size sliding window lekar uske andar calculations karna.

Matlab tum data ke chhote portions (jaise last 3 days, last 7 days) par aggregate functions

(mean, sum, max, etc.) apply karte ho.

Isse tumhe moving averages, rolling sums, aur short-term trends dekhne me madad milti hai.

📖 English:

Rolling windows perform calculations on a sliding window of data points.

👉 **7-day rolling mean** → Average of the last 7 days at each time point.

👉 **3-day rolling sum** → Sum of the last 3 values at each step.

Example:

👉 **df["sales"].rolling(7).mean()** → 7-day moving average.

👉 **df["temp"].rolling(3).sum()** → Rolling sum of last 3 values.

🗣️ Hinglish:

Socho tumhare paas ek **sliding window ka magnifying glass** hai 🔍.

👉 Tum ek time par **3 ya 7 values** dekhte ho aur unka average/sum nikalte ho.

👉 Jaise-jaise window aage slide hota hai, naye results aate rehte hain.

Isi ko hum **rolling calculation** kehte hain.

🔹 Syntax:

```
Rolling Mean
df["sales"].rolling(7).mean()


Rolling Sum
df["temp"].rolling(window=3).sum()


Rolling with custom function
df["values"].rolling(window=5).max()
```

• **Parameters / Notes:**

👉 **window** → Kitne size ka window (e.g., 3, 7, 30).

👉 **.mean()**, **.sum()**, **.max()**, **.min()** → Aggregation functions.

👉 **min_periods** → Minimum observations needed for calculation (default = window size).

- **Returns** → Rolling **DataFrame/Series** with calculated values.

📌 Real-Life Examples of Rolling Windows

🍎 1. 7-Day Moving Average of Sales

In [46]:

```
import pandas as pd
import numpy as np

dates = pd.date_range('2023-01-01', periods=15, freq="D")
df= pd.DataFrame({"Sales":np.random.randint(10,150,15)}, index=dates)
print(df)

print('\n\n',df['Sales'].rolling(7).mean())
```

```
            Sales
2023-01-01     71
2023-01-02    140
2023-01-03     69
2023-01-04     50
2023-01-05     58
2023-01-06     76
2023-01-07     98
2023-01-08     86
2023-01-09     16
2023-01-10     85
2023-01-11     97
2023-01-12     13
2023-01-13    102
2023-01-14     84
2023-01-15     44



 2023-01-01          NaN
2023-01-02          NaN
2023-01-03          NaN
2023-01-04          NaN
2023-01-05          NaN
2023-01-06          NaN
2023-01-07    80.285714
2023-01-08    82.428571
2023-01-09    64.714286
2023-01-10    67.000000
2023-01-11    73.714286
2023-01-12    67.285714
2023-01-13    71.000000
2023-01-14    69.000000
2023-01-15    63.000000
Freq: D, Name: Sales, dtype: float64
```

📝 Hinglish: Rozana sales data ka **7 din ka moving average** nikal liya.

🚗 2. 3-Day Rolling Sum of Temperature

In [47]:

```
df = pd.DataFrame({"Temp":[25,28,30,27,29,32,31]}, index=pd.date_range("2023-01-01", per
print(df['Temp'].rolling(3).sum())
```

```
2023-01-01     NaN
2023-01-02     NaN
2023-01-03    83.0
2023-01-04    85.0
2023-01-05    86.0
2023-01-06    88.0
2023-01-07    92.0
Freq: D, Name: Temp, dtype: float64
```

📝 Hinglish: Har din ke liye pichle 3 din ka **temperature ka total** calculate kiya.

📊 3. Rolling Maximum

In [48]:

```
df = pd.DataFrame({"Speed":[50,60,55,70,65,80,75]}, index=pd.date_range("2023-01-01", pe

print(df["Speed"].rolling(3).max())
```

```
2023-01-01     NaN
2023-01-02     NaN
2023-01-03    60.0
2023-01-04    70.0
2023-01-05    70.0
2023-01-06    80.0
2023-01-07    80.0
Freq: D, Name: Speed, dtype: float64
```

📝 Hinglish: Har din ke liye pichle 3 din me **sabse bada speed** dikhaya.

⚽ 4. Stock Price Rolling Average

In [49]:

```
df = pd.DataFrame({"Price":[100,102,104,106,108,110,115,120]}, index=pd.date_range("2023

print(df["Price"].rolling(window=5).mean())
```

```
2023-01-01      NaN
2023-01-02      NaN
2023-01-03      NaN
2023-01-04      NaN
2023-01-05    104.0
2023-01-06    106.0
2023-01-07    108.6
2023-01-08    111.8
Freq: D, Name: Price, dtype: float64
```

📝 Hinglish: Stock prices ka **5-day moving average** nikala jisse short-term trend smooth ho gaya.

💰 5. Real-Life: COVID Cases Rolling Average

In [50]:

```
df = pd.DataFrame({"Cases":[5,10,20,30,25,40,50,45,60,55]}, index=pd.date_range("2023-01

print(df["Cases"].rolling(7).mean())
```

```
2023-01-01          NaN
2023-01-02          NaN
2023-01-03          NaN
2023-01-04          NaN
2023-01-05          NaN
2023-01-06          NaN
2023-01-07    25.714286
2023-01-08    31.428571
2023-01-09    38.571429
2023-01-10    43.571429
Freq: D, Name: Cases, dtype: float64
```

📝 Hinglish: Rozana ke cases ko smooth karke **7-day rolling average** nikal liya (jaise news me dikhate hain).

11. □ Window Functions – Rolling, Expanding & Cumulative Ops

Operation / Category	Definition	Syntax / Example

Rolling Functions	Apply operations on a fixed-size moving window.	`df["sales"].rolling(3).m`
		`df["temp"].rolling(5).su`
		`df["sales"].rolling(3).m`
		`df["sales"].rolling(3).m`

Expanding Functions	Apply cumulative operations expanding from the start.	`df["sales"].expanding().`
		`df["marks"].expanding().`
		`df["sales"].expanding().`

Cumulative Functions	Perform cumulative calculations over entire series.	`df["sales"].cumsum()`
		`df["marks"].cummax()`
		`df["profit"].cummin()`
		`df["sales"].cumprod()`

Ranking	Rank values in a column (similar to SQL RANK).	`df["marks"].rank()`
		`df["sales"].rank(ascendi`
		`df["salary"].rank(methoc`

Shift / Lag / Lead	Shift values up/down in time (like SQL LAG/LEAD).	`df["sales"].shift(1) ← lag`
		`df["sales"].shift(-1) ← lead`

Operation / Category	Definition	Syntax / Example

Differencing	Find change between current and previous values.	`df["sales"].diff()`
		`df["sales"].diff(periods`

Percent Change	Calculate percentage change vs previous row.	`df["sales"].pct_change()`

Aggregate / Transform	Apply aggregate function over a partition (like SQL PARTITION BY).	`df.groupby("dept")["salary"].transform("su`
		`df.groupby("dept")["salary"].transform("me`
		`df.groupby("dept")["salary"].transform("mi`
		`df.groupby("dept")["salary"].transform("ma`
		`df.groupby("dept")["salary"].transform("co`

Cumulative / Running Aggregate	Apply cumulative aggregate per order (like SQL SUM()/AVG() OVER(ORDER BY …)).	`df["cum_sum"] = df["sales"].cumsum()`
		`df["cum_avg"] = df["sales"].expanding().`
		`df["cum_min"] = df["sales"].cummin()`
		`df["cum_max"] = df["sales"].cummax()`

Moving / Rolling Aggregate	Aggregate over a sliding window (like SQL ROWS BETWEEN …).	`df["roll_sum"] = df["sales"].rolling(3).s`
		`df["roll_avg"] = df["sales"].rolling(3).m`
		`df["roll_min"] = df["sales"].rolling(3).m`
		`df["roll_max"] = df["sales"].rolling(3).m`

Percent / Ratio / Ranking	Percentile rank, cumulative distribution, ratio to total.	`df["pct_rank"] = df["salary"].rank(pct=Tr`
		`df["cum_dist"] = df["salary"].rank(method / len(df)`
		`df["ratio"] = df["salary"] / df["salary"].sum()`

Operation / Category	Definition	Syntax / Example
Row Number	Sequential numbering per group.	`df["row_num"] = df.groupby("dept").cumco + 1`
Quartile / NTILE	Divide data into quantiles.	`df["quartile"] = pd.qcut(df["salary"], 4, labels=False)`

✅ Main Difference – Expanding vs Cumulative

Function	Definition / Usage	Example / Note
Expanding Functions	General-purpose cumulative calculation from start to current row. Tum har row pe koi bhi operation apply kar sakte ho (mean, sum, max, min, custom function).	`df["sales"].expanding().` Flexible → start se expand hota hai
Cumulative Functions	Special case of expanding. Predefined built-in operations only: cumsum, cummax, cummin, cumprod.	`df["sales"].cumsum()` Cumulative = expanding with fixed ops

🗣️ So basically: **cumulative = expanding with built-in fixed ops**, aur **expanding = flexible for any operation**.

📌 (i) What is Expanding Functions?

📏 **Definition — Expanding ka matlab hai cumulative calculation karna start se lekar current row tak.**
Matlab rolling ki tarah fixed window nahi hota, balki har naye step par calculation shuru se lekar ab tak hota hai.

📖 English:

Expanding functions perform cumulative calculations from the beginning of the dataset up to the current row.

👉 **Expanding mean** → Average from start till current point.

👉 **Expanding sum** → Total from beginning till now.

👉 **Expanding max** → Largest value so far.

🗣️ Hinglish:

Socho tum ek **notebook me roz ke marks** likh rahe ho 📒.

👉 Day 1 → bas 1st value ka mean.

👉 Day 2 → 1st + 2nd ka mean.

👉 Day 3 → 1st se 3rd tak ka mean.

Isi tarah expanding calculation hamesha **start se lekar ab tak** hota hai.

🔹 Syntax:

```
Expanding Mean
df["sales"].expanding().mean()


Expanding Sum
df["marks"].expanding().sum()


Expanding Max
df["values"].expanding().max()
```

- **Parameters / Notes**:

  👉 **min_periods** → Minimum observations required (default = 1).

  👉 **.mean()**, **.sum()**, **.max()**, **.min()** → Aggregation functions.

- **Returns** → Expanding **DataFrame/Series** with cumulative values.

📌 Real-Life Examples of Expanding Functions

🍎 1. Expanding Mean of Sales

In [1]:

```python
import pandas as pd

df = pd.DataFrame({'Sales': [10,20,30,40,50]})
print(df['Sales'].expanding().mean())
```

```
0    10.0
1    15.0
2    20.0
3    25.0
4    30.0
Name: Sales, dtype: float64
```

📝 Hinglish: Har step pe start se lekar ab tak ka average nikal liya.

🚗 2. Expanding Sum of Marks

In [2]:

```python
df = pd.DataFrame({"Marks":[50,60,70,80,90]})
print(df["Marks"].expanding().sum())
```

```
0     50.0
1    110.0
2    180.0
3    260.0
4    350.0
Name: Marks, dtype: float64
```

📝 Hinglish: Shuru se lekar har row tak ka cumulative total nikal liya.

📊 3. Expanding Maximum

In [3]:

```python
df = pd.DataFrame({"Speed":[40,55,50,70,65]})
print(df['Speed'].expanding().max())
```

```
0    40.0
1    55.0
2    55.0
3    70.0
4    70.0
Name: Speed, dtype: float64
```

📝 Hinglish: Har row tak ka maximum speed calculate kar diya.

⚽ 4. Expanding Min (Stock Prices)

In [4]:

```python
df = pd.DataFrame({"Price":[100,98,105,95,110]})
print(df['Price'].expanding().min())
```

```
0    100.0
1     98.0
2     98.0
3     95.0
4     95.0
Name: Price, dtype: float64
```

📝 Hinglish: Har din tak ka sabse chhota stock price dikhaya.

💰 5. Real-Life Example: Cumulative Sales

In [5]:

```
df = pd.DataFrame({"Sales":[200,300,400,500,600]})
df["Cumulative_Sales"] = df["Sales"].expanding().sum()
print(df)
```

```
   Sales  Cumulative_Sales
0    200             200.0
1    300             500.0
2    400             900.0
3    500            1400.0
4    600            2000.0
```

📝 Hinglish: Roz ke sales ka cumulative total nikal liya → jaise yearly sales ka progress dekhna.

📌 (ii) What is Cumulative Function?

📏 **Definition — Cumulative functions ka matlab hai poore series ke upar step-by-step cumulative calculation karna.**
Expanding jaisa hai, lekin yaha built-in cumulative methods directly provide kiye gaye hain jaise .cumsum(), .cummax(), .cummin(), .cumprod().

📖 English:

Cumulative functions compute a running total, running maximum, running minimum, or running product over the entire series.

🗣️ Hinglish:

Socho tumhare paas sales, marks, ya profit ka data hai. Tum bolte ho:

"Bhai, har row tak ka total de do" → **cumsum()**

"Ab tak ka maximum value bata do" → **cummax()**

"Ab tak ka minimum value bata do" → **cummin()**

"Sab ka product calculate kar do" → **cumprod()**

🔹 Syntax:

```
Cumulative Sum
df["sales"].cumsum()
```

```
Cumulative Maximum
df["marks"].cummax()


Cumulative Minimum
df["profit"].cummin()


Cumulative Product
df["sales"].cumprod()
```

- **Parameters / Notes**:

👉 **.cumsum(), .cummax(), .cummin(), .cumprod()** → Aggregation functions for cumulative calculations.

👉 Works directly on Series or DataFrame columns.

📌 Real-Life Examples of Cumulative Functions

🍎 1. Cumulative Sum of Sales

In [6]:

```python
import pandas as pd

df = pd.DataFrame({"Sales":[100,200,150,250,300]})
print(df["Sales"].cumsum())
```

```
0     100
1     300
2     450
3     700
4    1000
Name: Sales, dtype: int64
```

📝 Hinglish: Har step pe start se lekar current row tak ka total sales nikal liya.

🚗 2. Cumulative Maximum of Marks

In [7]:

```python
df = pd.DataFrame({"Marks":[50,60,55,70,65]})
print(df["Marks"].cummax())
```

```
0    50
1    60
2    60
3    70
4    70
Name: Marks, dtype: int64
```

📝 Hinglish: Har row tak ka ab tak ka highest marks calculate kar diya.

📊 3. Cumulative Minimum of Profit

In [8]:

```
df = pd.DataFrame({"Profit":[500,450,600,400,650]})
print(df["Profit"].cummin())
```

```
0    500
1    450
2    450
3    400
4    400
Name: Profit, dtype: int64
```

📝 Hinglish: Har step pe ab tak ka minimum profit dekh liya.

⚽ 4. Cumulative Product of Sales

In [9]:

```
df = pd.DataFrame({"Sales":[2,3,4,5]})
print(df["Sales"].cumprod())
```

```
0      2
1      6
2     24
3    120
Name: Sales, dtype: int64
```

📝 Hinglish: Har row tak ka sales product calculate kar diya (2 → 23 → 23*4 …).

💰 5. Real-Life Example: Stock Portfolio Growth

In [10]:

```
df = pd.DataFrame({"Return":[1.02,1.03,0.98,1.05,1.04]})
df["Cumulative_Return"] = df["Return"].cumprod()
print(df)
```

```
   Return  Cumulative_Return
0    1.02           1.020000
1    1.03           1.050600
2    0.98           1.029588
3    1.05           1.081067
4    1.04           1.124310
```

📝 Hinglish: Daily returns ko cumulative product me convert karke portfolio growth dikhaya.

📌 (iii) What is Ranking Function?

📏 **Definition — Ranking functions ka matlab hai ek column ke values ko order ya position assign karna, jaise SQL me RANK() function hota hai.**
Tum decide kar sakte ho ascending/descending order me rank chahiye aur tie-break kaise handle karna hai.

📖 English:

Ranking assigns ranks to values in a Series or DataFrame column. You can rank in ascending or descending order, and handle ties using different methods: average, min, max, dense, first.

🗣️ Hinglish:

Socho tumhare paas marks ya salary ka data hai. Tum bolte ho:

“Bhai, sabka rank bata do” → **rank()**
“Top scorer ko 1 rank de aur descending me order karo” → **rank(ascending=False)**
“Same marks wale log ka rank consecutive rakho” → **rank(method="dense")**

🔹 Syntax:

```
Default ranking (ascending)
df["marks"].rank()


Descending ranking
df["sales"].rank(ascending=False)


Dense ranking (tie wale consecutive rank)
df["salary"].rank(method="dense")


Other methods: "average", "min", "max", "first"
df["marks"].rank(method="first")
```

• **Parameters / Notes:**
👉 **ascending=True/False** → Rank ascending ya descending order me.
👉 **method** → Tie-break ka rule:
"average" → Tie ka average rank
"min" → Tie ko lowest rank assign kare
"max" → Tie ko highest rank assign kare
"dense" → Tie ke liye consecutive rank, gaps nahi
"first" → Row order ke hisaab se rank assign

📌 Real-Life Examples of Ranking Functions

🍎 1. Rank Marks in Ascending Order

In [14]:

```
import pandas as pd

df = pd.DataFrame({"Marks":[50,60,55,70,65]})
df["Rank"] = df["Marks"].rank()
print(df)
```

```
   Marks  Rank
0     50   1.0
1     60   3.0
2     55   2.0
3     70   5.0
4     65   4.0
```

📝 Hinglish: Marks ka ascending order me rank assign kar diya (lowest marks = rank 1).

🚗 2. Rank Sales in Descending Order

In [16]:

```
df = pd.DataFrame({"Sales":[200,450,300,450,500]})
df['Ranking'] = df["Sales"].rank(ascending=False)
print(df)
```

```
   Sales  Ranking
0    200      5.0
1    450      2.5
2    300      4.0
3    450      2.5
4    500      1.0
```

📝 Hinglish: Top sales = rank 1. Descending ranking use kiya.

📊 3. Dense Ranking of Salary

In [18]:

```
df = pd.DataFrame({"Salary":[50000,60000,60000,700000,45000]})
print(df["Salary"].rank(method="dense"))
```

```
0    2.0
1    3.0
2    3.0
3    4.0
4    1.0
Name: Salary, dtype: float64
```

📝 Hinglish: Tie wale salaries ko consecutive rank diya, gaps nahi bane.

⚽ 4. First Method Ranking

In [20]:

```python
df = pd.DataFrame({"Marks":[80,80,75,90]})
df['First_Rank'] = df["Marks"].rank(method="first")
print(df)
```

```
   Marks  First_Rank
0     80         2.0
1     80         3.0
2     75         1.0
3     90         4.0
```

📝 Hinglish: Tie wale marks ko row order ke hisaab se rank assign kiya.

💰 5. Real-Life Example: Student Scores Ranking

In [21]:

```python
df = pd.DataFrame({
    "Student":["Alice","Bob","Charlie","David"],
    "Score":[85,90,85,95]
})
df["Rank"] = df["Score"].rank(ascending=False, method="dense")
print(df)
```

```
   Student  Score  Rank
0    Alice     85   3.0
1      Bob     90   2.0
2  Charlie     85   3.0
3    David     95   1.0
```

📝 Hinglish: Highest scorer ko 1 rank diya aur tie wale ko consecutive rank assign kiya.

📌 (iv) What is Shift / Lag / Lead Function?

📏 **Definition — Shift / Lag / Lead functions ka matlab hai ek column ke values ko upar ya neeche move karna, jaise SQL me LAG()/LEAD() function hota hai.**
Ye time-series analysis aur previous/next row comparison ke liye bohot useful hai.

📖 English:

The **.shift()** function shifts values in a Series or DataFrame by a specified number of periods.
Positive values shift down (lag), negative values shift up (lead).

🗣️ Hinglish:

Socho tumhare paas daily sales ya stock prices ka data hai. Tum bolte ho:

“Bhai, kal ka value abhi ke row me dikhao” → **lag (shift(1))**
“Next day ka value abhi ke row me dikhao” → **lead (shift(-1))**

🔹 Syntax:

```
Lag: shift values down
df["sales"].shift(1)


Lead: shift values up
df["sales"].shift(-1)


Multiple periods
df["sales"].shift(2)   # 2 rows down
df["sales"].shift(-3)  # 3 rows up


Shift with fill value
df["sales"].shift(1, fill_value=0)
```

- **Parameters / Notes**:

  👉 **periods** → Kitne rows shift karne hain (default = 1)

  👉 **fill_value** → Missing positions me kaunsa value bharna hai (default = NaN)

  👉 Positive periods → Lag (down)

  👉 Negative periods → Lead (up)

📌 Real-Life Examples of Shift / Lag / Lead Functions

🍎 1. Lag Example: Previous Day Sales

In [28]:

```
import pandas as pd

df = pd.DataFrame({"Sales":[100,200,150,300,250]})
df["Prev_Day_Sales"] = df['Sales'].shift(1).round(0).fillna(0).astype('int')
print(df)
```

```
   Sales  Prev_Day_Sales
0    100               0
1    200             100
2    150             200
3    300             150
4    250             300
```

📝 Hinglish: Har row me previous day ka sales value add kar diya (lag).

🚗 2. Lead Example: Next Day Sales

In [30]:

```
df['Next_Day_Sale'] = df['Sales'].shift(-1).fillna(0).astype('int')
print(df)
```

```
   Sales  Prev_Day_Sales  Next_Day_Sale
0    100               0            200
1    200             100            150
2    150             200            300
3    300             150            250
4    250             300              0
```

📝 Hinglish: Har row me next day ka sales value add kar diya (lead).

📊 3. Multiple Period Shift

In [31]:

```
df["Two_Days_Ago"] = df["Sales"].shift(2)
print(df)
```

```
   Sales  Prev_Day_Sales  Next_Day_Sale  Two_Days_Ago
0    100               0            200           NaN
1    200             100            150           NaN
2    150             200            300         100.0
3    300             150            250         200.0
4    250             300              0         150.0
```

📝 Hinglish: 2 din pehle ka sales har row me dikha diya.

⚽ 4. Shift with Fill Value

In [32]:

```
df["Prev_Day_Sales_Zero"] = df["Sales"].shift(1, fill_value=0)
print(df)
```

```
   Sales  Prev_Day_Sales  Next_Day_Sale  Two_Days_Ago  Prev_Day_Sales_Zero
0    100               0            200           NaN                    0
1    200             100            150           NaN                  100
2    150             200            300         100.0                  200
3    300             150            250         200.0                  150
4    250             300              0         150.0                  300
```

📝 Hinglish: Missing lag value ko 0 se replace kar diya.

💰 5. Real-Life Example: Stock Prices Change

In [33]:

```
df = pd.DataFrame({"Price":[100,105,110,120,125]})
df["Prev_Price"] = df["Price"].shift(1)
df["Price_Change"] = df["Price"] - df["Prev_Price"]
print(df)
```

```
   Price  Prev_Price  Price_Change
0    100         NaN           NaN
1    105       100.0           5.0
2    110       105.0           5.0
3    120       110.0          10.0
4    125       120.0           5.0
```

📝 Hinglish: Previous day price nikal ke current price se subtract kiya → daily change calculate ho gaya.

⚡ Bhai, shift vs lag vs lead subtle difference:
👉 **Shift** = `.shift()` function (general).
👉 **Lag** = shift down (`shift(1)`), previous row values.
👉 **Lead** = shift up (`shift(-1)`), next row values.

📌 (v) What is Differencing Function?

📏 **Definition — Differencing function ka matlab hai current row aur previous row ke beech ka difference nikalna.**
Ye time-series me trend ya change detect karne ke liye bohot useful hai.

📖 English:

The **.diff()** function calculates the difference between a current value and its previous value (or nth previous value). It's widely used in time-series analysis to compute growth, returns, or daily changes.

🗣️ Hinglish:

Socho tumhare paas daily sales ya stock prices ka data hai. Tum bolte ho:

"Bhai, har din ka change bata do previous day se" → **.diff()**
"2 din pehle ke comparison ka change bhi bata do" → **.diff(periods=2)**

🔹 Syntax:

```
Difference with previous row (default periods=1)
df["sales"].diff()


Difference with 2 previous rows
df["sales"].diff(periods=2)


Multiple periods
df["profit"].diff(3)   # Current row - 3 rows before
```

```
Negative periods (future difference)
df["sales"].diff(-1)
```

- **Parameters / Notes:**

  👉 **periods** → Number of rows to calculate difference with (default = 1)

  👉 Positive periods → Previous row(s) difference

  👉 Negative periods → Future row(s) difference

📌 Real-Life Examples of Differencing Functions

🍎 1. Daily Sales Difference

In [34]:

```
import pandas as pd

df = pd.DataFrame({"Sales":[100,200,150,300,250]})
df["Daily_Change"] = df["Sales"].diff()
print(df)
```

```
   Sales  Daily_Change
0    100           NaN
1    200         100.0
2    150         -50.0
3    300         150.0
4    250         -50.0
```

📝 Hinglish: Har row me previous day se sales ka difference nikal liya.

🚗 2. 2-Day Difference

In [35]:

```
df["Two_Day_Change"] = df["Sales"].diff(periods=2)
print(df)
```

```
   Sales  Daily_Change  Two_Day_Change
0    100           NaN             NaN
1    200         100.0             NaN
2    150         -50.0            50.0
3    300         150.0           100.0
4    250         -50.0           100.0
```

📝 Hinglish: Har row me 2 din pehle se difference calculate kiya.

📊 3. Negative Period Difference (Lead Style)

In [36]:

```
df["Next_Day_Change"] = df["Sales"].diff(-1)
print(df)
```

```
   Sales  Daily_Change  Two_Day_Change  Next_Day_Change
0    100           NaN             NaN           -100.0
1    200         100.0             NaN             50.0
2    150         -50.0            50.0           -150.0
3    300         150.0           100.0             50.0
4    250         -50.0           100.0              NaN
```

📝 Hinglish: Current row - next row ka difference (future comparison).

⚽ 4. Real-Life Example: Stock Returns

In [37]:

```
df = pd.DataFrame({"Price":[100,105,110,120,125]})
df["Price_Change"] = df["Price"].diff()
df["Return_%"] = df["Price"].diff() / df["Price"].shift(1) * 100
print(df)
```

```
   Price  Price_Change  Return_%
0    100           NaN       NaN
1    105           5.0  5.000000
2    110           5.0  4.761905
3    120          10.0  9.090909
4    125           5.0  4.166667
```

📝 Hinglish: Stock price ka daily change aur percentage return calculate kar diya.

💰 5. Real-Life Example: Temperature Change

In [38]:

```
df = pd.DataFrame({"Temp":[25,28,30,27,29]})
df["Temp_Change"] = df["Temp"].diff()
print(df)
```

```
   Temp  Temp_Change
0    25          NaN
1    28          3.0
2    30          2.0
3    27         -3.0
4    29          2.0
```

📝 Hinglish: Har din ka temperature change nikal liya → trend analyze karne ke liye.

⚡ Bhai, shift vs differencing subtle difference:
👉 **Shift** = Row values ko move karna (lag/lead), difference calculate nahi karta.
👉 **Differencing** = Row ke values ka difference nikalta hai, automatically current - previous (ya nth row).

📌 (vi) What is Percent Change Function?

📏 Definition — Percent change function ka matlab hai current row aur previous row ke beech ka percentage change nikalna.
Ye time-series me growth rate ya returns calculate karne ke liye bohot useful hai.

📖 English:

The **.pct_change()** function calculates the percentage change between the current value and its previous value (or nth previous value). Commonly used in finance, sales growth analysis, or any sequential data.

🗣️ Hinglish:

Socho tumhare paas daily sales ya stock prices ka data hai. Tum bolte ho:

“Bhai, har din ka growth % bata do previous day se” → **.pct_change()**
“2 din pehle ke comparison ka % change bhi bata do” → **.pct_change(periods=2)**

🔹 Syntax:

```
Percent change with previous row (default periods=1)
df["sales"].pct_change()

Percent change with 2 previous rows
df["sales"].pct_change(periods=2)

Multiply by 100 to convert to percentage
df["sales"].pct_change() * 100

Negative periods (future % change)
df["sales"].pct_change(-1)
```

• **Parameters / Notes**:

👉 **periods** → Number of rows to calculate change with (default = 1)
👉 Positive periods → Previous row(s) percentage change
👉 Negative periods → Future row(s) percentage change

📌 Real-Life Examples of Percent Change Functions

🍎 1. Daily Sales Percent Change

In [43]:

```python
import pandas as pd

df = pd.DataFrame({"Sales":[100,200,150,300,250]})
df["Daily_%_Change"] = df["Sales"].pct_change() * 100
df = df.fillna(0).astype('int')
print(df)
```

```
   Sales  Daily_%_Change
0    100               0
1    200             100
2    150             -25
3    300             100
4    250             -16
```

📝 Hinglish: Har row me previous day se sales ka percentage change nikal liya.

🚗 2. 2-Day Percent Change

In [44]:

```python
df["Two_Day_%_Change"] = df["Sales"].pct_change(periods=2) * 100
print(df)
```

```
   Sales  Daily_%_Change  Two_Day_%_Change
0    100               0               NaN
1    200             100               NaN
2    150             -25         50.000000
3    300             100         50.000000
4    250             -16         66.666667
```

📝 Hinglish: Har row me 2 din pehle ke comparison ka % change calculate kiya.

📊 3. Negative Period Percent Change (Lead Style)

In [45]:

```python
df["Next_Day_%_Change"] = df["Sales"].pct_change(-1) * 100
print(df)
```

```
   Sales  Daily_%_Change  Two_Day_%_Change  Next_Day_%_Change
0    100               0               NaN         -50.000000
1    200             100               NaN          33.333333
2    150             -25         50.000000         -50.000000
3    300             100         50.000000          20.000000
4    250             -16         66.666667                NaN
```

📝 Hinglish: Current row - next row ka percentage change (future comparison).

⚽ 4. Real-Life Example: Stock Returnsm

In [47]:

```python
df = pd.DataFrame({"Price":[100,105,110,120,125]})
df["Return_%"] = df["Price"].pct_change() * 100
print(df)
```

```
   Price  Return_%
0    100       NaN
1    105  5.000000
2    110  4.761905
```

```
3     120   9.090909
4     125   4.166667
```

📝 Hinglish: Stock price ka daily % return calculate kar diya → finance me commonly use hota hai.

💰 **5. Real-Life Example: Temperature Change %**

In [48]:

```
df = pd.DataFrame({"Temp":[25,28,30,27,29]})
df["Temp_%_Change"] = df["Temp"].pct_change() * 100
print(df)
```

```
   Temp  Temp_%_Change
0     25            NaN
1     28      12.000000
2     30       7.142857
3     27     -10.000000
4     29       7.407407
```

📝 Hinglish: Har din ka temperature % change nikal liya → trends aur anomalies detect karne ke liye.

⚡ Bhai, differencing vs percent change subtle difference:
👉 **Differencing** = Current - Previous (absolute change).
👉 **Percent Change** = ((Current - Previous)/Previous) * 100 → growth rate.

📌 (vii) What is Aggregate / Transform Function?

📏 **Definition — Aggregate / Transform functions ka matlab hai ek column ke values par group-wise operation apply karna, jaise SQL me PARTITION BY hota hai.**
Tum har group ke liye sum, mean, min, max, count, etc. nikal sakte ho aur original DataFrame me har row ke saath attach kar sakte ho.

📖 English:

The **.transform()** function applies an aggregation function over a group and returns a Series with the same size as the original. Useful when you want group-wise calculations but keep the original row alignment.

🗣️ Hinglish:

Socho tumhare paas employees ka salary data aur department ka column hai. Tum bolte ho:

“Har employee ke liye department ka total salary add kar do” → **.transform("sum")**
“Har employee ke liye department ka average salary chahiye” → **.transform("mean")**

Ye original DataFrame ka shape same rakhta hai, bas har row me group calculation add ho jata hai.

- 🔹 Syntax:

```
Sum per group
df.groupby("dept")["salary"].transform("sum")


Mean per group
df.groupby("dept")["salary"].transform("mean")


Minimum per group
df.groupby("dept")["salary"].transform("min")


Maximum per group
df.groupby("dept")["salary"].transform("max")


Count per group
df.groupby("dept")["salary"].transform("count")
```

• **Parameters / Notes**:

👉 **Function** → "sum", "mean", "min", "max", "count", ya custom function

👉 Works on a **grouped object** (groupby)

👉 **Returns a Series** aligned with the original DataFrame

📌 Real-Life Examples of Aggregate / Transform Functions

🍎 1. Sum of Salary per Department

In [55]:

```
import pandas as pd

df = pd.DataFrame({
    "Employee":["A","B","C","D","E"],
    "Dept":["HR","IT","HR","IT","HR"],
    "Salary":[50000,60000,55000,65000,52000]
})
```

```
df["Dept_Total_Salary"] = df.groupby("Dept")["Salary"].transform("sum")
print(df)
```

```
  Employee Dept  Salary  Dept_Total_Salary
0        A   HR   50000             157000
1        B   IT   60000             125000
2        C   HR   55000             157000
3        D   IT   65000             125000
4        E   HR   52000             157000
```

📝 Hinglish: Har employee ke liye uske department ka total salary calculate kar diya.

🚗 2. Average Salary per Department

In [50]:

```
df["Dept_Avg_Salary"] = df.groupby("Dept")["Salary"].transform("mean")
print(df)
```

```
  Employee Dept  Salary  Dept_Total_Salary  Dept_Avg_Salary
0        A   HR   50000             157000     52333.333333
1        B   IT   60000             125000     62500.000000
2        C   HR   55000             157000     52333.333333
3        D   IT   65000             125000     62500.000000
4        E   HR   52000             157000     52333.333333
```

📝 Hinglish: Har employee ke row me uske department ka average salary add kar diya.

📊 3. Minimum and Maximum Salary per Department

In [52]:

```
df["Dept_Min_Salary"] = df.groupby("Dept")["Salary"].transform("min")
df["Dept_Max_Salary"] = df.groupby("Dept")["Salary"].transform("max")
print(df)
```

```
  Employee Dept  Salary  Dept_Total_Salary  Dept_Avg_Salary  Dept_Min_Salary  Dept_Max_S
alary
0        A   HR   50000             157000     52333.333333            50000
55000
1        B   IT   60000             125000     62500.000000            60000
65000
2        C   HR   55000             157000     52333.333333            50000
55000
3        D   IT   65000             125000     62500.000000            60000
65000
4        E   HR   52000             157000     52333.333333            50000
55000
```

📝 Hinglish: Har department ka lowest aur highest salary har employee ke row me add ho gaya.

⚽ 4. Count of Employees per Department

In [56]:

```
import pandas as pd

df = pd.DataFrame({
    "Employee":["A","B","C","D","E"],
    "Dept":["HR","IT","HR","IT","HR"],
    "Salary":[50000,60000,55000,65000,52000]
})
```

```
df["Dept_Count"] = df.groupby("Dept")["Salary"].transform("count")
print(df)
```

```
  Employee Dept  Salary  Dept_Count
0        A   HR   50000           3
1        B   IT   60000           2
2        C   HR   55000           3
3        D   IT   65000           2
4        E   HR   52000           3
```

📝 Hinglish: Har employee ke row me uske department ka total employees count attach kar diya.

💰 5. Real-Life Example: Sales per Region

In [57]:

```
df = pd.DataFrame({
    "Region":["East","West","East","West","East"],
    "Sales":[1000,2000,1500,2500,1200]
})
df["Region_Total_Sales"] = df.groupby("Region")["Sales"].transform("sum")
print(df)
```

```
  Region  Sales  Region_Total_Sales
0   East   1000                3700
1   West   2000                4500
2   East   1500                3700
3   West   2500                4500
4   East   1200                3700
```

📝 Hinglish: Har sales record ke row me us region ka total sales calculate kar diya.

⚡ Bhai, aggregate vs transform subtle difference:
👉 **Aggregate** `(agg)` → Returns one row per group.
👉 **Transform** `(transform)` → Returns same size as original, aligned with rows, useful for per-row group calculations.

📌 (viii) What is Percent / Ratio / Ranking Function?

📏 **Definition — Percent / Ratio / Ranking functions ka matlab hota hai data me proportions, percentile rank, ya ratio to total calculate karna. Ye analysis me use hota hai jab hume dekhna ho ki ek value total ke respect me kitna contribute karti hai, ya uska rank distribution me kya position hai.**

📖 English:

.rank(pct=True) → Gives percentile rank (value’s relative standing between 0 and 1).

.rank(method="first") / len(df) → Used for cumulative distribution.

value / total → Simple way to calculate ratio to total.

🗣️ Hinglish:

Socho tumhare paas employees ki salary ka data hai. Tum bolte ho:

“Bhai, har employee ka percentile rank de do salary ke basis par” → **.rank(pct=True)**

“Har employee cumulative distribution me kitni position pe hai?” → **.rank(method="first") / len(df)**

“Bhai, salary ka kitna hissa total salary me contribute karta hai?” → **salary / salary.sum()**

🔹 Syntax:

```
Percentile rank
df["pct_rank"] = df["salary"].rank(pct=True)

Cumulative distribution
df["cum_dist"] = df["salary"].rank(method="first") / len(df)

Ratio to total
df["ratio"] = df["salary"] / df["salary"].sum()
```

• **Parameters / Notes**:

👉 **pct=True** → Percentile rank return karega (0 to 1)

👉 **method** → Tie-breaking ka rule (first, average, min, max, dense)

👉 **/ len(df)** → Normalization ke liye use hota hai cumulative distribution banane ke liye

📌 Real-Life Examples of Percent / Ratio / Ranking

🍎 1. Percentile Rank of Salary

In [1]:

```python
import pandas as pd

df = pd.DataFrame({
    "Employee":["A","B","C","D","E"],
    "Salary":[50000,60000,55000,65000,52000]
})
df["Percentile_Rank"] = df["Salary"].rank(pct=True)
print(df)
```

```
  Employee  Salary  Percentile_Rank
0        A   50000              0.2
```

```
1         B   60000                0.8
2         C   55000                0.6
3         D   65000                1.0
4         E   52000                0.4
```

📝 Hinglish: Salary ke basis pe har employee ka percentile rank (0 se 1 ke beech) calculate kar diya.

🚗 2. Cumulative Distribution of Salary

In [2]:

```
df["Cum_Dist"] = df["Salary"].rank(method="first") / len(df)
print(df)
```

```
  Employee  Salary  Percentile_Rank  Cum_Dist
0        A   50000              0.2       0.2
1        B   60000              0.8       0.8
2        C   55000              0.6       0.6
3        D   65000              1.0       1.0
4        E   52000              0.4       0.4
```

📝 Hinglish: Salary ke values ko order karke cumulative distribution nikal diya.

📊 3. Ratio of Salary to Total Salary

In [3]:

```
df["Ratio_to_Total"] = df["Salary"] / df["Salary"].sum()
print(df)
```

```
  Employee  Salary  Percentile_Rank  Cum_Dist  Ratio_to_Total
0        A   50000              0.2       0.2        0.177305
1        B   60000              0.8       0.8        0.212766
2        C   55000              0.6       0.6        0.195035
3        D   65000              1.0       1.0        0.230496
4        E   52000              0.4       0.4        0.184397
```

📝 Hinglish: Har employee ki salary ka contribution total salary me kitna % hai, wo nikal diya.

⚽ 4. Real-Life Example: Sales Percentile Rank

In [4]:

```
df = pd.DataFrame({
    "Region":["East","West","East","West","North"],
    "Sales":[1000,2000,1500,2500,1200]
})
df["Sales_Rank_Pct"] = df["Sales"].rank(pct=True)
print(df)
```

```
  Region  Sales  Sales_Rank_Pct
0   East   1000             0.2
1   West   2000             0.8
2   East   1500             0.6
3   West   2500             1.0
4  North   1200             0.4
```

📝 Hinglish: Har region ke sales ko percentile rank me convert kar diya.

💰 5. Real-Life Example: Market Share (Ratio to Total)

In [5]:

```
df["Market_Share"] = df["Sales"] / df["Sales"].sum() * 100
print(df)
```

```
  Region  Sales  Sales_Rank_Pct  Market_Share
0   East   1000             0.2     12.195122
1   West   2000             0.8     24.390244
2   East   1500             0.6     18.292683
3   West   2500             1.0     30.487805
4  North   1200             0.4     14.634146
```

📝 Hinglish: Har region ka sales percentage nikal ke market share calculate kar diya.

⚡ Bhai, **Percent / Ratio / Ranking vs Normal Ranking subtle difference**:
👉 Ranking (normal) = order ke hisaab se rank assign karta hai (1,2,3…).
👉 Percentile Rank = relative standing (0 to 1).
👉 Ratio = value ka contribution total me.
👉 Cumulative Distribution = dataset ke andar progressive position.

📌 (ix) What is Row Number?

📏 **Definition — Row Number ka matlab hota hai ek group ke andar sequential numbering dena, jaise SQL me ROW_NUMBER() OVER (PARTITION BY ...) hota hai.**
Ye har group ke andar rows ko 1 se start karke numbering assign karta hai.

📖 English:

The **.cumcount()** function counts rows within each group starting from 0.
Agar tum +1 kar doge, toh numbering 1 se start hogi (SQL row number style).

🗣️ Hinglish:

Socho tumhare paas employees ka data hai department ke hisaab se. Tum bolte ho:

"Har department ke andar har employee ko ek row number de do sequential order me."

👉 Ye bilkul SQL wale **ROW_NUMBER()** ka Pandas version hai.

- 🔹 Syntax:

```
Row number per group
df["row_num"] = df.groupby("dept").cumcount() + 1
```

- **Parameters / Notes**:

👉 **groupby("col")** → jis column ke hisaab se grouping karni hai

👉 **.cumcount()** → har group ke andar counting karega (0 se start)

👉 **+1** → numbering ko 1 se start karne ke liye

👉 **Returns** → Ek naya column with sequential row number per group

📌 Real-Life Examples of Row Number

🍎 1. Basic Row Number per Department

In [6]:

```
import pandas as pd

df = pd.DataFrame({
    "Employee":["A","B","C","D","E","F"],
    "Dept":["HR","IT","HR","IT","HR","IT"]
})

df["Row_Num"] = df.groupby("Dept").cumcount() + 1
print(df)
```

```
  Employee Dept  Row_Num
0        A   HR        1
1        B   IT        1
2        C   HR        2
3        D   IT        2
4        E   HR        3
5        F   IT        3
```

📝 Hinglish: Har department ke andar employees ko 1,2,3... numbering mil gayi.

🚗 2. Row Number after Sorting

In [7]:

```
df = df.sort_values(["Dept","Employee"])
df["Row_Num"] = df.groupby("Dept").cumcount() + 1
print(df)
```

```
  Employee Dept  Row_Num
0        A   HR        1
2        C   HR        2
4        E   HR        3
1        B   IT        1
3        D   IT        2
5        F   IT        3
```

📝 Hinglish: Sorting karne ke baad row numbers bhi naye order ke hisaab se assign ho gaye.

📊 3. Real-Life Example: Sales per Region

In [10]:

```
df = pd.DataFrame({
    "Region":["East","East","West","West","West", "East"],
    "Sales":[1000,1500,2000,2500,2700,1200]
})

df["Row_Num"] = df.groupby("Region").cumcount() + 1
print(df)
```

```
  Region  Sales  Row_Num
0   East   1000        1
1   East   1500        2
2   West   2000        1
3   West   2500        2
4   West   2700        3
5   East   1200        3
```

📝 Hinglish: Har region ke sales records ko sequential row number assign ho gaya.

⚽ 4. Real-Life Example: Employee Promotions

In [11]:

```
df = pd.DataFrame({
    "Dept":["HR","HR","IT","IT","IT"],
    "Employee":["Raj","Simran","Aman","Neha","Vikram"],
    "Year":[2021,2022,2021,2022,2023]
})

df["Row_Num"] = df.groupby("Dept").cumcount() + 1
print(df)
```

```
  Dept Employee  Year  Row_Num
0   HR      Raj  2021        1
1   HR   Simran  2022        2
2   IT     Aman  2021        1
3   IT     Neha  2022        2
4   IT   Vikram  2023        3
```

📝 Hinglish: Har department ke andar promotions ko year-wise row numbers assign kar diya.

📌 (x) What is Quartile / NTILE Function?

📏 **Definition — Quartile / NTILE function ka matlab hai data ko equal-sized buckets (quantiles) me tod dena.**
Jaise tum ek dataset ko 4 parts me (quartiles), 10 parts me (deciles), ya 100 parts me (percentiles) split kar sakte ho. Ye distribution analysis ke liye helpful hota hai.

📖 English:

The **pd.qcut()** function divides data into equal-sized quantile bins. Example: **q=4** → data 4 quartiles me split hoga. Labels automatically assign hote hain ya tum custom labels de sakte ho.

🗣️ Hinglish:

Socho tumhare paas employees ka salary data hai. Tum bolte ho:

“Bhai, salary ko 4 groups (quartiles) me divide kar do” → **pd.qcut(..., 4)**

“Bhai, mujhe 10 groups (deciles) chahiye” → **pd.qcut(..., 10)**

🔹 Syntax:

```
Quartile binning (4 groups)
df["quartile"] = pd.qcut(df["salary"], 4, labels=False)


NTILE (custom n groups)
df["ntile"] = pd.qcut(df["salary"], q=10, labels=False)
```

• **Parameters / Notes**:

👉 **q** → Number of quantiles (4 = quartile, 10 = decile, 100 = percentile)

👉 **labels=False** → Assigns integer labels (0..n-1)

👉 **duplicates="drop"** → Removes duplicate edges if bins can’t be divided properly

👉 **Returns** → A column with quantile bin assignment

📌 Real-Life Examples of Quartile / NTILE

🍎 1. Quartiles of Salary

In [13]:

```
import pandas as pd

df = pd.DataFrame({
    "Employee":["A","B","C","D","E","F","G","H"],
    "Salary":[30000,40000,50000,60000,70000,80000,90000,100000]
})

df["Quartile"] = pd.qcut(df["Salary"], 4, labels=False)
print(df)
```

```
  Employee  Salary  Quartile
0        A   30000         0
1        B   40000         0
2        C   50000         1
3        D   60000         1
4        E   70000         2
```

```
5        F   80000        2
6        G   90000        3
7        H  100000        3
```

📝 Hinglish: Salary ko 4 equal buckets me tod diya (0=lowest quartile, 3=highest quartile).

🚗 2. Deciles (NTILE=10)

In [14]:

```
df["Decile"] = pd.qcut(df["Salary"], 10, labels=False)
print(df)
```

```
  Employee  Salary  Quartile  Decile
0        A   30000         0       0
1        B   40000         0       1
2        C   50000         1       2
3        D   60000         1       4
4        E   70000         2       5
5        F   80000         2       7
6        G   90000         3       8
7        H  100000         3       9
```

📝 Hinglish: Salary ko 10 groups (deciles) me divide kar diya.

📊 3. Real-Life Example: Sales Quartile Analysis

In [15]:

```
df = pd.DataFrame({
    "Region":["East","West","North","South","East","West","North","South"],
    "Sales":[1200,3000,1500,7000,2200,4500,1800,9000]
})

df["Sales_Quartile"] = pd.qcut(df["Sales"], 4, labels=["Q1","Q2","Q3","Q4"])
print(df)
```

```
  Region  Sales Sales_Quartile
0   East   1200             Q1
1   West   3000             Q3
2  North   1500             Q1
3  South   7000             Q4
4   East   2200             Q2
5   West   4500             Q3
6  North   1800             Q2
7  South   9000             Q4
```

📝 Hinglish: Har region ka sales data ko Q1 (lowest) se Q4 (highest) quartile me classify kar diya.

⚽ 4. Market Segmentation (Customer Spend Quartiles)

In [16]:

```
df = pd.DataFrame({
    "Customer":["C1","C2","C3","C4","C5","C6"],
    "Spend":[500,2000,1500,3000,1000,4000]
})

df["Spend_Group"] = pd.qcut(df["Spend"], 3, labels=["Low","Medium","High"])
print(df)
```

```
  Customer  Spend Spend_Group
0       C1    500         Low
1       C2   2000      Medium
2       C3   1500      Medium
3       C4   3000        High
4       C5   1000         Low
5       C6   4000        High
```

📝 Hinglish: Customers ko unke kharch ke hisaab se Low, Medium, High groups me baant diya.

📌 What is Pandas Profiling?

📏 **Definition — Pandas Profiling ek library hai jo automated Exploratory Data Analysis (EDA) report generate karti hai.**
Report me dataset ka summary, missing values, data types, distributions, correlations, duplicates — sab kuch ek hi jagah pe milta hai.

📖 English:

The **pandas_profiling.ProfileReport** creates an interactive HTML report for any DataFrame.
Super useful for quickly understanding datasets before analysis or modeling.

🗣️ Hinglish:

Socho tumhare paas ek bada dataset hai aur tum bolte ho:

“Bhai, mujhe is data ka full x-ray chahiye — summary, graphs, missing values sab ek report me”
→ **Pandas Profiling karega**.

🔹 Syntax:

```
from pandas_profiling import ProfileReport


profile = ProfileReport(df)

profile.to_file("report.html")
```

- **Parameters / Notes**:

👉 **explorative=True** → Detailed mode me aur zyada stats deta hai

👉 **minimal=True** → Fast report with limited features

👉 **title="My Report"** → Custom title add karne ke liye

👉 **correlations** → Control correlation methods (pearson, spearman, kendall)

👉 **Returns** → An interactive HTML EDA report

🙏 Thank You for Being Part of the Pandas Journey!

From zero to data wizard—your growth proves that consistency beats complexity. 🌱

Remember: Every dataset has a story, and you now have the power to tell it. 📊

Keep practicing, keep experimenting, and never stop being curious—the magic lies in the details. 🔍

Your journey doesn't end here—it's just the beginning of turning raw data into real-world impact. 🚀

Stay motivated, stay consistent, and remember: Wizards aren't born, they're made—one dataset at a time. ✨

In [ ]: