PANDAS

Pandas

- Pandas is a popular tool (library) in Python that helps you work with data easily.
- Pandas is a powerful Python library used for data analysis and manipulation.

Why is it used?

- It helps you organize data in tables (like Excel sheets).
- You can quickly read, write, and change data.
- It makes it easy to analyse and find patterns or information from data.
- You can clean messy data and prepare it for reports or graphs.
- In short, Pandas is used to handle and analyse data in a simple and fast way.

It provides two main data structures:

1. Series – 1D Labelled Array

- In Pandas, a Series is a one-dimensional array-like object that can hold data of any type (integers, floats, strings, etc.) along with an associated index, which labels each element.
- Think of it as a single column in a spreadsheet or a labelled array.
- A Series is like a single column of data. Imagine a list of numbers, names, or dates, but with labels for each item.
- In list we have to search for the position of number manually but in series there will be some index with the help of that we can access the element.
- And also index in series gives the context to data instead of just numbers.

Key points about Series:

- It is **one-dimensional** — just like a list or a column in Excel.
- Each item in a Series has an **index** (a label or number) to identify it. If you don't give labels, Pandas gives numbers starting from 0 automatically.
- You can store many types of data: numbers, words, dates, etc.
- It can be created from a Python list, dictionary, or array.

What is a Pandas Series?

- A **Series** is a one-dimensional data structure in Pandas with two main components:
  - **Data**: The actual values (can be numbers, strings, or other types).
  - **Index**: A sequence of labels that uniquely identify each element in the Series (default is 0-based integer index).
- It's similar to a NumPy array but with the added feature of customizable indices.

Key Concepts to Understand About Pandas Series

To ace an interview, you should be familiar with the following aspects of a Series:

1. Creating a Series

You can create a Series in multiple ways:

- **From a list**:

python Collapse Wrap Run Copy

```
s = pd.Series([1, 2, 3, 4])
```

- **From a dictionary** (keys become the index):

```python
s = pd.Series({'a': 1, 'b': 2, 'c': 3})
```

- **From a scalar value** (with an index):

```python
s = pd.Series(5, index=[0, 1, 2])
```

- **From a NumPy array**:

```python
import numpy as np
s = pd.Series(np.array([1, 2, 3]))
```

- **Key Points**:
  - You can specify a custom index (e.g., strings or dates).
  - The `dtype` is automatically inferred but can be explicitly set (e.g., `pd.Series([1, 2, 3], dtype='float64')`).

2. Attributes of a Series

Know these common attributes to access metadata:

- `s.values`: Returns the underlying data as a NumPy array.
- `s.index`: Returns the index of the Series.
- `s.dtype`: Returns the data type of the values.
- `s.shape`: Returns the shape (e.g., `(n,)` for n elements).
- `s.size`: Returns the number of elements.
- `s.name`: The name of the Series (useful when converting to DataFrame).

```python
s = pd.Series([1, 2, 3], index=['a', 'b', 'c'], name='Numbers')
print(s.values)  # Output: [1 2 3]
print(s.index)   # Output: Index(['a', 'b', 'c'], dtype='object')
print(s.name)    # Output: Numbers
```

3. Indexing and Selecting Data

- **Access by index label**:

```python
s = pd.Series([10, 20, 30], index=['x', 'y', 'z'])
print(s['x'])  # Output: 10
```

- **Access by position (integer-based)**:

```python
print(s.iloc[0])  # Output: 10
```

- **Slicing**:

```python
print(s['x':'y'])  # Output: x    10
                   #         y    20
                   # dtype: int64
```

- **Boolean indexing:**

```python
print(s[s > 15])  # Output: y    20
                  #         z    30
                  # dtype: int64
```

7. Converting Series

- To **DataFrame**: `s.to_frame()`.
- To **list**: `s.tolist()`.
- To **NumPy array**: `s.values` or `s.to_numpy()`.
- To **dictionary**: `s.to_dict()`.

---

2. Data Frame – 2D Labelled Data Structure (like a table)

What is a Data Frame?

A Data Frame is like a full table with rows and columns — like an Excel sheet or a SQL table.
A Data Frame is like a table in Excel — it has rows and columns. Each column can store a different type of data, like numbers, text, or dates. In Python, we often use Pandas Data Frame to store and work with structured data so we can filter, sort, and analyse it easily

Key points about Data Frame:

- It is **two-dimensional** — it has rows and columns.
- Each column is a **Series** (so a DataFrame is made of many Series).
- Both rows and columns have labels (called index for rows and column names for columns).
- You can store different types of data in different columns (numbers in one column, text in another). Think of it as a collection of multiple Series (columns) that share the same row index.
- Each column in a Data Frame is a Series.
- You can perform powerful operations like filtering, sorting, grouping, etc.

Example:

```python
import pandas as pd

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

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

```
      Name  Age      City
0    Alice   25  New York
1      Bob   30    London
2  Charlie   35     Paris
```

Creating Data Frames:

1. Using Dictionary of List:

```python
import pandas as pd

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

# Create DataFrame
df = pd.DataFrame(data)

# Display DataFrame
print(df)
```

---

2. Using List of Dictionary

```python
import pandas as pd

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

# Create DataFrame
df = pd.DataFrame(data)

# Display DataFrame
print(df)
```

---

1. Reading from CSV file

```python
import pandas as pd

df = pd.read_csv('data.csv')  # 'data.csv' is the file name
print(df)
```

- **CSV (Comma-Separated Values)** is the most common format for storing tabular data.
- `pd.read_csv()` automatically converts the file into a DataFrame.

---

2. Reading from Excel file

```python
df = pd.read_excel('data.xlsx', sheet_name='Sheet1')
print(df)
```

- Excel files can have multiple sheets; use `sheet_name` to choose one.
- Requires **openpyxl** or **xlrd** library.

Reading JSON Directly from a File (Normal JSON File)

This is used when your JSON file is a **dictionary** or **list of dictionaries**.

Example JSON file → `data.json`

```json
[
    {"name": "A", "age": 20, "city": "Delhi"},
    {"name": "B", "age": 25, "city": "Mumbai"},
    {"name": "C", "age": 30, "city": "Chennai"}
]
```

Code to Read It

```python
import pandas as pd

df = pd.read_json("data.json")
print(df)
```

Output

name	age	city
A	20	Delhi
B	25	Mumbai
C	30	Chennai

Reading Line-by-Line JSON (JSON Lines Format)

Also called **JSONL** or **NDJSON**

Each line is a separate JSON object.

Example file → `data.jsonl`

```json
{"name": "A", "age": 20, "city": "Delhi"}
{"name": "B", "age": 25, "city": "Mumbai"}
{"name": "C", "age": 30, "city": "Chennai"}
```

Code to Read It

```python
import pandas as pd

df = pd.read_json("data.jsonl", lines=True)
print(df)
```

Why use `lines=True`?

Because each line is an independent JSON object, not a list.

Handling Nested JSON

Example JSON

```json
{
    "id": 1,
    "name": "Raksha",
    "address": {
        "city": "Udupi",
        "pincode": 576101
    }
}
```

This is difficult to read normally.

Solution → json_normalize()

```python
import json
import pandas as pd

data = {
    "id": 1,
    "name": "Raksha",
    "address": {
        "city": "Udupi",
        "pincode": 576101
    }
}

df = pd.json_normalize(data)
print(df)
```

Output

id	name	address.city	address.pincode
1	Raksha	Udupi	576101

json_normalize makes nested JSON flat, which is perfect for analysis.

Reading From SQL Database:

To read any table from a SQL database (MySQL, PostgreSQL, SQLite, etc.) into a Pandas DataFrame, we use SQLAlchemy + pandas.read_sql().

Install Required Libraries:

For MySQL

```nginx
pip install sqlalchemy
pip install pymysql
```

For PostgreSQL

```nginx
pip install sqlalchemy
pip install psycopg2
```

Import and create Database Connection

```python
from sqlalchemy import create_engine
import pandas as pd
```

MySQL Connection

```python
engine = create_engine("mysql+pymysql://username:password@localhost:3306/database_name")
```

PostgreSQL Connection

```python
engine = create_engine("postgresql+psycopg2://username:password@localhost:5432/database_name")
```

> Replace **username**, **password**, **host**, and **database_name** as per your system.

Read Table from Database

There are **two main ways**.

A. Using a SQL Query (Most Common)

```python
query = "SELECT * FROM students"
df = pd.read_sql(query, engine)
print(df)
```

Use this method when:

- You want only selected columns
- You want rows with a filter (WHERE)
- You want to join tables

Example:

```python
df = pd.read_sql("SELECT name, age FROM students WHERE age > 20", engine)
```

B. Read Entire Table Without Writing SQL

```python
df = pd.read_sql_table("students", engine)
print(df)
```

Use this when:

- You want the whole table
- You don't want to write SQL manually

Example: Reading Two Tables

Let's say your database contains:

- **students** table
- **marks** table

Read them:

```python
df_students = pd.read_sql("SELECT * FROM students", engine)
df_marks = pd.read_sql("SELECT * FROM marks", engine)
```

Or using `read_sql_table()`:

```python
df_students = pd.read_sql_table("students", engine)
df_marks = pd.read_sql_table("marks", engine)
```

Notes:

- read_sql_query() and read_sql() are almost the same; read_sql_query() is for queries, read_sql_table() is for entire tables.
- Using SQLAlchemy is recommended because it works with multiple databases.

Writing to Different Files:

1. **Writing to CSV File:**

- **df.to_csv("Hello",index=False)**

2. **Writing to Excel File:**

- **df.to_excel("Hello",index=False)**

Attributes

1. **To check top rows**
- **df.head()**
- By default, top 5 rows.

2. **To check bottom rows**
- **df.tail()**
- By default, bottom 5 rows.

3. **To check random rows**
- **df.sample()**
- By default, random 1 rows.

4. **To check column name, data type and number of non-null values**
- **df.info()**

5. **To check statistical summary**
- **df.describe()** → **Gives only for numerical columns**
- **df.describe(include='all')** → **For all columns**

6. **To get number of rows and columns**
   df.shape

7. **To get all columns**
   df.columns

8. **To check datatype of each column**
   df.dtypes

9. **To check total number of elements**
   df.size

10. **To check number of dimensions**
    df.ndim

Selection and Indexing

Example to Understand:

	ID	Name	Age	Department	Salary	Joining_Date	Performance_Score
0	1	Person1	25	HR	50000	2020-01-01	3
1	2	Person2	30	IT	60000	2020-12-31	4
2	3	Person3	22	Finance	55000	2021-12-31	5
3	4	Person4	28	IT	62000	2022-12-31	3
4	5	Person5	35	HR	58000	2023-12-31	4
5	6	Person6	40	Finance	72000	2024-12-30	5
6	7	Person7	27	IT	64000	2025-12-30	4
7	8	Person8	33	HR	50000	2026-12-30	3
8	9	Person9	29	Finance	68000	2027-12-30	5
9	10	Person10	31	IT	70000	2028-12-29	4

1. To Select Columns

2 Ways:

a) To select single column

df[“column_name’]

Example: df[“ID”]

- Use single Square brackets

b) To Select Multiple Columns

df[“column_name1, “column_name2”.....]

Example: df[[“ID”,”Name”,”Age”,”Department”]]

- Use double brackets otherwise it will show an error

---

2. To Select Rows

By Label:

```
df.loc[2]          # Row with index label 2
```

By Position:

```
df.iloc[2]         # Third row
```

By Condition:

```
df[df['Age'] > 30]     # All rows where Age > 30
```

Common Mistakes:

```
df['Age'] > 30    # ✅ Returns Boolean Series
df['Age' > 30]    # ❌ Wrong syntax
```

.loc and .iloc Concept

1. The Core Difference

Think of a Pandas Data Frame like an Excel sheet with:

- Rows → with a number label on the left
- Columns → with names at the top

`.loc` → Location by NAME (label-based)

- You tell Pandas **the name (label)** of the row or column you want.
- Works with **index labels** for rows and **column names** for columns.

`.iloc` → Location by POSITION (number-based)

- You tell Pandas **the number position** of the row or column (starting at 0).
- Works just like list indexing in Python.

1. `.loc`

`.loc` is used for **label-based selection** in Pandas.

That means:

- You **use names (labels)** for rows and columns, not numbers.
- It works with the **index labels** of rows and the **column names**.

Syntax:

```
df.loc[row_labels, column_labels]
```

- `row_labels` → name(s) of the row(s) you want.
- `column_labels` → name(s) of the column(s) you want.

Examples 1: df.loc[2]
Returns all columns for the row with label 1
Example 2: df.loc[0,"Name"]
Row with label 0, column 'Name' → "John"
Example 3: df.loc[0:2]
Returns rows with labels 0, 1, and 2 (inclusive!)
Example 4:

```
df.loc[:, 'Name']
# All rows, only the Name column

df.loc[:, ['Name', 'Age']]
# All rows, only Name and Age columns
```

---

2. `.iloc`

`.iloc` is used for **integer position-based selection** in Pandas.

That means:

- You **use numbers (positions)** for rows and columns.
- It ignores row/column names and looks at **where** they are.

Syntax:

```
df.iloc[row_positions, column_positions]
```

- `row_positions` → position(s) of the row(s) you want (starting from 0).
- `column_positions` → position(s) of the column(s) you want (starting from 0).

Example 1:

```
df.iloc[1]
# Returns all columns for the row at position 1 → Alice's row
```

Example 2:

```
df.iloc[0:2]
# Returns rows at positions 0 and 1 (exclusive of position 2)
```

Example 3:

```
df.iloc[0, 1]
# Row at position 0, column at position 1 → "John"
```

Example 4:

```
df.iloc[:, 1]
# All rows, only the column at position 1 (Name column)

df.iloc[:, [1, 2]]
# All rows, columns at positions 1 and 2 (Name & Age)
```

Example 5:

```
df.iloc[0:2, 1:3]
# Rows 0 and 1, columns 1 and 2 (Name & Age)
```

Boolean Indexing

- Boolean indexing is a way to filter rows or columns in a Data Frame or Series based on True/False conditions.
- Think of it like saying:
  "Show me only the rows where this condition is True."

2. The Core Steps (The 20% That Covers 80% Use-Cases)

Step 1 – Create a Condition

- A condition returns a **Boolean Series** (True or False for each row).

```python
import pandas as pd

data = pd.DataFrame({
    'Name': ['Alice', 'Bob', 'Charlie', 'David'],
    'Age': [25, 30, 35, 40],
    'Score': [85, 60, 75, 90]
})

condition = data['Age'] > 30
print(condition)
```

Output:

```yaml
0    False
1    False
2     True
3     True
Name: Age, dtype: bool
```

Step 2 – Apply the Condition to Filter

```python
data[condition]
# OR directly:
data[data['Age'] > 30]
```

Result:

```markdown
     Name  Age  Score
2  Charlie   35     75
3    David   40     90
```

Step 3 – Combine Multiple Conditions

- Use `&` for AND, `|` for OR, and `~` for NOT.
- **IMPORTANT**: Wrap each condition in parentheses `()`.

```python
# Age > 30 AND Score > 80
data[(data['Age'] > 30) & (data['Score'] > 80)]

# Age < 35 OR Score < 70
data[(data['Age'] < 35) | (data['Score'] < 70)]

# NOT Age > 30
data[~(data['Age'] > 30)]
```

Step 5 – Boolean Indexing with `.isin()` and `.str`

For checking multiple values:

```python
data[data['Name'].isin(['Alice', 'David'])]
```

For string conditions:

```python
data[data['Name'].str.contains('a', case=False)]
```

Issue	Why It Happens	How to Fix
1. Missing parentheses in multiple conditions	Python's & and `	` have lower precedence than comparison operators.
2. Using and/or instead of &/`	`**	and/or work for single values, not Series.
3. Comparing with NaN	NaN is never equal to anything, even itself.	Use .isna() or .notna()
4. Chain indexing warning	Doing df[df['Age'] > 30]['Name'] = ... can cause SettingWithCopyWarning.	Use .loc when assigning values.
5. Forgetting == for equality	Writing df['Age'] = 30 will overwrite the column instead of comparing.	Use == for comparisons.

Quick Cheat Sheet:

```python
# Simple filter
df[df['col'] > value]

# Multiple conditions
df[(df['col1'] > value1) & (df['col2'] < value2)]

# NOT condition
df[~(df['col'] == value)]

# Multiple matches
df[df['col'].isin(['A', 'B'])]

# String matching
df[df['col'].str.contains('pattern', case=False)]

# Missing values
df[df['col'].isna()]
df[df['col'].notna()]
```

✅ In Pandas (`str.contains`)

Pandas uses **regex**, not SQL wildcards.

What you want	SQL	Pandas regex
starts with S	`S%`	`^S`
ends with S	`%S`	`S$`
contains S	`%S%`	`S`
S followed by anything	`S%`	`S.*`

---

unique ()

- unique () returns the unique values from a pandas Series (or column of a Data Frame) as a NumPy array.
- It works only on a series, not on a directly whole data frame.

Example:

```
import pandas as pd

df = pd.DataFrame({
    'City': ['Delhi', 'Mumbai', 'Delhi', 'Chennai', 'Mumbai', 'Delhi']
})

df['City'].unique()
```

Output:

```
python

array(['Delhi', 'Mumbai', 'Chennai'], dtype=object)
```

- Shows each city **once**, in the **order they first appear**.

Edge Cases:

1. **Work with Nan:**
   - NaN is considered a **unique value**.
   - All NaN values are treated as **one**.

```
python

import numpy as np

df = pd.DataFrame({'A': [1, 2, np.nan, 2, np.nan]})
df['A'].unique()
# Output: array([1., 2., nan])
```

2. **Works on categorical and object dtype.**
   - For **categorical** columns, it's faster because categories are stored efficiently.
3. **Works only on Series, not entire DataFrame.**

In pandas

- Axis = 0 → Top to Bottom
- Axis = 1 → Left to Right

---

nunique ()

- nunique () returns the number of unique values in a pandas Series or Data Frame.
- Think of it like: “Count how many different values I have.”
  - **Example:**

```
import pandas as pd
import numpy as np

df = pd.DataFrame({
    'City': ['Delhi', 'Mumbai', 'Delhi', 'Chennai', 'Mumbai', np.nan],
    'Population': [100, 200, 100, 150, 200, np.nan]
})

df['City'].nunique()
# Output: 3
```

Explanation:

- The unique cities are `Delhi, Mumbai, Chennai`.
- NaN is **ignored** by default (`dropna=True`).

Edge Cases:

1. **While Counting number of unique values it ignores null values.**
2. **Using on a Series or Data frame**
   Example: For Series :

- df[“Age”].nunique() → Gives the single value
- For Data frame df.nunique() → Gives the unique value for each column

3. **Changing axis for rows**

- axis=0 → count per column (default)
- axis=1 → count per row

```
df.nunique(axis=1)
```

Counts how many unique values each row has.

value_counts()

- value_counts() returns the **unique values** in a Series along with how many times each value appears.
- Think of it like: “Give me a frequency table for this column

Example:

```
import pandas as pd
import numpy as np

df = pd.DataFrame({
    'City': ['Delhi', 'Mumbai', 'Delhi', 'Chennai', 'Mumbai', np.nan]
})

df['City'].value_counts()
```

Output:

```
yaml

Delhi      2
Mumbai     2
Chennai    1
Name: City, dtype: int64
```

- Shows **each unique value** and its **count**.
- NaN is **ignored** by default.

Edge Cases to Remember:

```
pd.Series(['delhi', 'Delhi']).value_counts()
# 'delhi' and 'Delhi' are different
```

- Numbers `1` and `1.0` are treated as **the same**.
- NaN ignored unless `dropna=False`.
- Sorting by value instead of count:

```
df
```

	Name	Department	Score
0	Raksha	Data	88
1	Akash	Finance	92
2	Raksha	Data	88
3	Sahana	HR	75
4	Manoj	IT	90
5	Akash	Finance	92

```
df.value_counts()
```

```
Name    Department  Score
Akash   Finance     92       2
Raksha  Data        88       2
Manoj   IT          90       1
Sahana  HR          75       1
Name: count, dtype: int64
```

- Value_counts for whole df gives how many times each record appears.
- That is unique record (Rows) and count of it.

Handling Duplicates

- **Find Duplicates:** duplicated ()
- **Remove Duplicates:** drop_duplicates()

1. Find Duplicates

	Department	Salary	Employee_ID	Experience_Years	Gender	Joining_Date	Performance_Rating
0	HR	3000	E001	1	F	2020-05-10	3
1	IT	4000	E002	3	M	2019-08-15	4
2	IT	4500	E003	5	M	2021-03-20	5
3	Finance	5000	E004	7	F	2018-11-05	2
4	HR	3500	E005	2	F	2020-07-22	3
5	Finance	4800	E006	4	M	2019-09-12	4
6	IT	4200	E007	6	M	2022-01-10	5
7	HR	3700	E008	3	F	2021-12-01	3

1. For Whole Data

```
df.duplicated()
```

```
0    False
1    False
2    False
3    False
4    False
5    False
6    False
7    False
dtype: bool
```

➔ To get Duplicate records that is duplicate of Previous Records

```
df[df.duplicated()]
```

2. For Specific Column

```
df[df.duplicated("Department")]
```

	Department	Salary	Employee_ID	Experience_Years	Gender	Joining_Date	Performance_Rating
2	IT	4500	E003	5	M	2021-03-20	5
4	HR	3500	E005	2	F	2020-07-22	3
5	Finance	4800	E006	4	M	2019-09-12	4
6	IT	4200	E007	6	M	2022-01-10	5
7	HR	3700	E008	3	F	2021-12-01	3

```
df[df.duplicated(subset=["age","gender"])]
```

	transactions_id	sale_date	sale_time	customer_id	gender	age	category
3	1180	2022-01-06	08:53:00	85	Male	41	Clothing
4	1522	2022-11-14	08:35:00	hoi	Male	46	Beauty
5	1559	2022-08-20	07:40:00	49	Female	40	Clothing

2. Drop Duplicates

- **Based on all Column**

```
df.drop_duplicates()
```

	Department	Salary	Employee_ID	Experience_Years	Gender	Joining_Date	Performance_Rating
0	HR	3000	E001	1	F	2020-05-10	3
1	IT	4000	E002	3	M	2019-08-15	4
2	IT	4500	E003	5	M	2021-03-20	5
3	Finance	5000	E004	7	F	2018-11-05	2
4	HR	3500	E005	2	F	2020-07-22	3
5	Finance	4800	E006	4	M	2019-09-12	4
6	IT	4200	E007	6	M	2022-01-10	5
7	HR	3700	E008	3	F	2021-12-01	3

- **Based on one specific Column**

```
df.drop_duplicates("Department")
```

	Department	Salary	Employee_ID	Experience_Years	Gender	Joining_Date	Performance_Rating
0	HR	3000	E001	1	F	2020-05-10	3
1	IT	4000	E002	3	M	2019-08-15	4
3	Finance	5000	E004	7	F	2018-11-05	2

- **Based on Multiple Column**

```
df.drop_duplicates(["Department","Gender"])
```

	Department	Salary	Employee_ID	Experience_Years	Gender	Joining_Date	Performance_Rating
0	HR	3000	E001	1	F	2020-05-10	3
1	IT	4000	E002	3	M	2019-08-15	4
3	Finance	5000	E004	7	F	2018-11-05	2
5	Finance	4800	E006	4	M	2019-09-12	4

🔑 **Extra small things you should know (just to be interview-safe):**

- **Keep parameter**

```python
df.duplicated(keep="last")     # Marks first as duplicate, keeps last
df.duplicated(keep=False)      # Marks all duplicates as True
df.drop_duplicates(keep="last")   # Keeps the last occurrence
df.drop_duplicates(keep=False)    # Removes *all* duplicates
```

👉 Useful when interviewer asks: *"What if I don't want to keep the first duplicate, but the last one?"*

- **In-place operation**

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

👉 Saves memory if you don't want a copy.

astype () Function

- astype() **changes the data type** of one or more pandas columns (or entire DataFrame) to a new type you specify.

Syntax:

```python
df['column'] = df['column'].astype(new_type)
```

Examples:

Change column to integer	`df['Age'] = df['Age'].astype(int)`
Change column to float	`df['Price'] = df['Price'].astype(float)`
Change column to string	`df['ID'] = df['ID'].astype(str)`
Change column to category	`df['Gender'] = df['Gender'].astype('category')`
Change multiple columns	`df = df.astype({'Age': 'float', 'Gender': 'category'})`

For Date:

```
df['Date'] = pd.to_datetime(df['Date'])
```

Edge Case:

1. **Invalid Conversion (ValueError)**
   - You can't convert `"abc"` to `int`.

```python
pd.Series(['1', '2', 'abc']).astype(int)  # ❌ Error
```

Renaming Columns

- In Pandas rename () function is used to rename the columns in the data frame

1. To Change All columns

```
df.columns = ['New_column1', 'New_column2', 'New_column3', ...]
```

2. To Change Specific Column Names

```
df.rename(columns={"Old_Column1": "New_Column1", "Old_Column2": "New_Column2"}, inplace=True)
```

Edge Cases:

1. **inplace=True means no return value**

- If you use inplace=True, pandas **doesn't return a new DataFrame**.

```
df = df.rename(columns={'old':'new'})  # ✅ Works (returns new DF)
df.rename(columns={'old':'new'}, inplace=True)  # ❌ No return, changes original
```

Handling Inconsistent Data Entry

1. Why inconsistent data entry matters

- As a data analyst, even perfect-looking datasets may have hidden issues.

```
"India"
" india"
"INDIA"
"India "
```

These **look similar** to us but are **different values** for Python/Pandas.

If you try to count values, sort, or merge data, they'll be treated as separate items.

2. Common problems you'll face

- **Extra spaces** → " India " vs "India"
- **Different case styles** → "india" vs "India" vs "INDIA"
- **Mixed whitespace** → tabs \t, newlines \n
- **Hidden characters** → non-breaking spaces \xa0
- **Inconsistent spelling** → "Bangalore" vs "Bengaluru"
- **Empty cells that are actually spaces** → " " instead of NaN

Different Techniques

1. Removing extra spaces

- Removes Leading or Trailing Spaces – Spaces at the beginning or end of the text.

```
df['Country'] = df['Country'].str.strip()
```

- Normalise Multiple Spaces to Single

```
df['Country'] = df['Country'].str.replace(r'\s+', ' ', regex=True)
```

{**Note:** We have to give str. strip () or etc because .**strip ()** is a method of a Python string }

Breaking it down

1. `df['Country']` → Select the column you want to clean.
2. `.str.replace(...)` → String operation applied element-wise to each cell.
3. `r'\s+'`
   - `\s` → means *any whitespace character* (space, tab, newline, etc.)
   - `+` → means *one or more times*
   - Together: match *any run of whitespace*, no matter how long.
4. `' '` → The replacement: a single space.
5. `regex=True` → Tells Pandas the pattern is a **regular expression**, not a literal string.

Handling case sensitivity

```
df['Country'] = df['Country'].str.lower()   # everything to lowercase
# OR
df['Country'] = df['Country'].str.title()   # First letter capital
```

- Choose a standard format to the whole dataset

3. **Removing hidden/non-breaking spaces**

   Sometimes copy-pasted data from Excel/Web has weird spaces (\xa0):

```
df['Country'] = df['Country'].str.replace(u'\xa0', ' ')
df['Country'] = df['Country'].str.strip()
```

4. **Standardizing spelling**

```
replace_map = {
    'Bangalore': 'Bengaluru',
    'Bangaluru': 'Bengaluru'
}
df['City'] = df['City'].replace(replace_map)
```

```
df["Country"] = df["Country"].str.replace(r"Bangalore|Bangaluru", "Bengaluru", regex=True)
```

- Useful when you have a fixed known – variations.

5. **Handling empty or whitespace-only values**

```
df['Country'] = df['Country'].replace(r'^\s*$', None, regex=True)
df['Country'] = df['Country'].fillna('Unknown')  # optional
```

- This converts “blank but not really blank” to actual NaN (or None).

Some Methods

1. **For converting all values into uppercase letters**

```
df["Product"]=df["Product"].str.upper()
```

2. **For converting all values into lowercase letters**

```
df["Product"]=df["Product"].str.lower()
```

3. **For converting all values into Title case letters**

```
df["Product"]=df["Product"].str.title()
```

4. **To remove hidden leading (Spaces at the right end) and trailing spaces (Spaces at the left end)**

```
df["Quantity"] = df["Quantity"].str.strip()
```

5. **To remove hidden trailing spaces (Spaces at the left end)**

```
df["Quantity"] = df["Quantity"].str.lstrip()
```

6. **To remove hidden leading spaces (Spaces at the right end)**

```
df["Quantity"] = df["Quantity"].str.rstrip()
```

7. **Inconsistent Categories**

- When categorical columns have similar but slightly different values:

```
"Male", "male", "M", "m"
"New York", "NY", "new york"
```

Fix:

```
python

df['Gender'] = df['Gender'].str.lower().replace({'m': 'male', 'f': 'female'})
```

Finding Missing Values

Missing Values - A place in your dataset where a value should be there, but it's not - if it is empty, unknown, or unavailable.

1. First, why missing values matter

If you don't handle missing data properly:

- Your calculations may be wrong (e.g., averages become inaccurate).
- Some ML models or charts will fail.
- You might miss important patterns.

To Check for Null Values

```
df.isnull()   # --> Gives True in the dataset wherever the data is null
```

	Customer	Product	Quantity	Price	Date
0	False	False	False	False	False
1	False	False	False	False	False
2	False	False	False	False	False

```
df.isnull().sum()     # --> Gives count of null values in each column of the dataframe
```

```
Customer    0
Product     0
Quantity    0
Price       0
Date        0
dtype: int64
```

```
df.isnull().sum().sum()    # --> Gives count of null values in whole dataframe
```

```
0
```

```
df.notnull()   # --> Gives True in the dataset wherever the data is not null
```

	Customer	Product	Quantity	Price	Date
0	True	True	True	True	True
1	True	True	True	True	True
2	True	True	True	True	True
3	True	True	True	True	True
4	True	True	True	True	True
5	True	True	True	True	True

```
df.notnull().sum()  # --> Gives count of non null values in each column of the dataframe
```

```
Customer    6
Product     6
Quantity    6
Price       6
Date        6
dtype: int64
```

```
df.notnull().sum().sum()  # --> Gives count of non null values in whole dataframe
```

```
30
```

Handling Missing Values

1. Filling with Constant Values

```
df['Column'] = df['Column'].fillna(0)          # Fill with number
df['City'] = df['City'].fillna('Unknown')      # Fill with string
```

When to use:

- When a missing value actually means “none” or “not applicable”.
- Example: Product discount missing → probably means 0% discount.
- Example: City missing in customer info → use "Unknown" as placeholder.

Edge cases:

- Don’t fill with a constant just to “get rid of nulls” — it can add wrong information.
- If you fill numeric columns with 0, remember that 0 might affect averages and sums.

---

2. Forward Fill (ffill)

```
df['Column'] = df['Column'].fillna(method='ffill')
```

What it does: Takes the value from the **previous row** and fills the missing spot.

When to use:

- Time-series or ordered data where the **previous value logically continues**.
- Example: Stock prices missing for a day — you assume it stayed the same as the day before.

Edge cases:

- If the first value in the column is missing, forward fill won’t work for it (there’s no previous value).
- Be careful in data where **values change often** — forward fill might give wrong info.

---

3. Backward Fill (bfill)

```
df['Column'] = df['Column'].fillna(method='bfill')
```

When to use:

- Similar to forward fill, but when **future value** is a better assumption.
- Example: Sensor reading missing at 2 PM → use 3 PM’s reading if it makes sense.

Edge cases:

- If last value is missing, backward fill won’t work.
- May create unrealistic jumps if values vary a lot.

---

4. Filling with Mean, Median, Mode

```
df['Column'] = df['Column'].fillna(df['Column'].mean())   # Mean
df['Column'] = df['Column'].fillna(df['Column'].median()) # Median
df['Column'] = df['Column'].fillna(df['Column'].mode()[0])# Mode
```

When to use:

- **Mean:** Data is normally distributed (no extreme outliers).
    - Example: Average age of customers.

- **Median:** Data has **outliers**.
    - Example: Income data — a billionaire customer will skew the mean.
- **Mode:** Categorical columns where most common value makes sense.
    - Example: Filling missing gender with most common gender in dataset.

Edge cases:

- Filling with mean/median can **hide variability** and make data look cleaner than it really is.
- Don't use mean/median for categorical text columns (they don't work logically).

---

5. Dropping Null Values

```
df.dropna(axis=0)  # Drop rows with any nulls
df.dropna(axis=1)  # Drop columns with any nulls
```

When to use:

- When only a few rows are missing and they don't represent an important group.
- When a column has too many missing values (e.g., >50%) and isn't worth keeping.

Edge cases:

- Dropping can lead to loss of valuable data if many rows/columns are removed.
- For small datasets, losing data might be worse than filling.

---

6. Replacing Null Values (More Control)

```
df['Column'] = df['Column'].replace(np.nan, 'CustomValue')
```

When to use:

- Similar to fillna, but works well when replacing multiple different "null" indicators (like "N/A", "None", 0, etc.) in one go.

It is used to replace normal values as well.

For an example,

In one particular column wherever 6 is there if you want to replace it with 7.

df14 = df.replace(to_replace = 6, value =7)

---

7. Interpolation

```
df['Column'] = df['Column'].interpolate(method='linear')
```

What it does: Fills missing values by estimating between known values.

- Example: If values are 10 (Day 1) and 14 (Day 3), interpolation fills Day 2 with 12.

When to use:

- Continuous numerical data where values change gradually.
- Example: Temperature readings, stock prices, scientific measurements.

Edge cases:

- Won't work well if data has sudden jumps.
- Doesn't make sense for categorical columns.

---

3. The 20% Knowledge That Covers 80% of Cases

If you remember only this decision flow, you'll handle most cases well:

1. Check % of missing values.
- If too high (>50%) in a column → drop the column.
- If very low (<5% rows) → consider dropping those rows.
2. Understand the column type & meaning.
1. Categorical: Use mode or a constant like "Unknown".
2. Numerical:
   - Normal distribution? → Mean.
   - Skewed or has outliers? → Median.
   - Gradual change (time-series)? → Interpolation / forward fill.
3. Preserve data meaning.
   - Don't just "fill to remove nulls" — think if the fill value makes sense in real life.

---

4. Common Edge Cases

- Null values in ID columns → usually should be dropped, IDs must be unique and complete.
- All values in a column missing → drop column (no way to fill meaningfully).
- Mixed missing indicators ('NA', 'null', 0, '') → first standardize them to NaN using replace () before filling.
- Data where order matters (time-series) → prefer forward/backward fill or interpolation over mean/median.

"There are several ways to handle missing values in Pandas, and I choose the method based on the type of data and how much is missing.

For example:

- If the missing value actually means 'none' or 'not applicable', I fill it with a constant value like 0 or 'Unknown'.
- In time-series data, I use forward fill (ffill) or backward fill (bfill) because the previous or next value logically continues.
- For numerical columns, if data is normally distributed, I use mean; if it has outliers, I prefer median; and for categorical data, I use mode.
- If too many values are missing (like more than 50%), I may drop the column completely.
- And if only a few rows have nulls, I might drop those rows instead.
- For continuous data that changes smoothly, like temperature or stock prices, I use interpolation to estimate the missing value.

The main point is — I never just fill nulls blindly. I first check what the column represents and whether filling or dropping makes logical sense."

To check How much % values in a column is null

```
Few missing rows (1-5%) → drop rows
Many missing rows → fill
Column mostly missing (>50%) → drop column
```

```
print(f"{(df.isnull().sum() / len(df)) * 100}")
```

- Here, `len(df))` gives total number of rows.

% of Missing Values	Action
< 5%	Drop the rows (small impact)
5% – 50%	Fill using mean, median, mode, or constant
>50%	Drop the column (too much missing data)

Outliers

An **outlier** is a value in your data that **is very different** from most of the other values.

Finding Outliers:

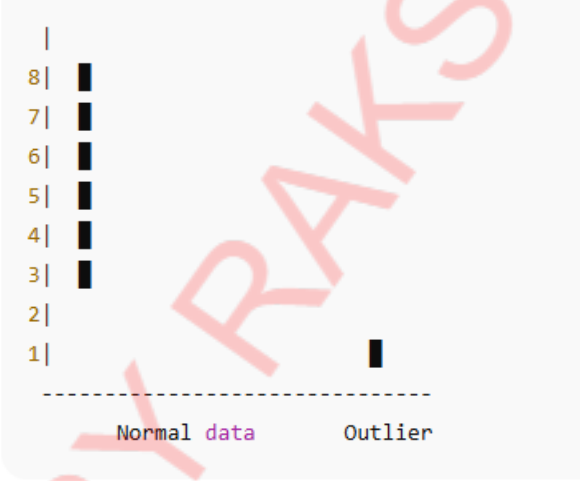
1 Using the IQR (Interquartile Range) Method

```
import pandas as pd

# Sample data
data = {'Age': [15, 16, 15, 16, 15, 80]}
df = pd.DataFrame(data)

# Calculate Q1 and Q3
Q1 = df['Age'].quantile(0.25)
Q3 = df['Age'].quantile(0.75)
IQR = Q3 - Q1

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

# Define outliers
outliers = df[(df['Age'] < (lower_bound)) | (df['Age'] > (upper_bound))]
print(outliers)
```

2 Using Histogram

- A histogram is a graph that shows the distribution of data using bars.
- The x-axis represents value ranges (called bins).
- The y-axis shows the frequency (how many times values appear) in each bin.
- You can think of it as grouping data into ranges and seeing how it is spread.

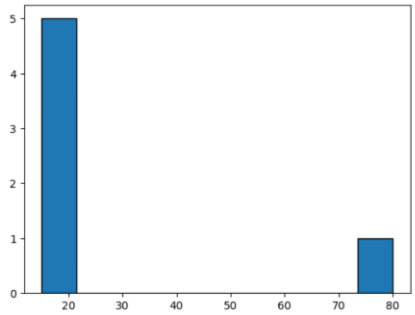
Suppose We Have Data Like: [18, 20, 21, 22, 22, 23, 23, 24, 100]



The bar at 100 is very far away → **possible outlier**.

How to Use:

```
import matplotlib.pyplot as plt
plt.hist(df["Age"], bins=10, edgecolor='black')
```

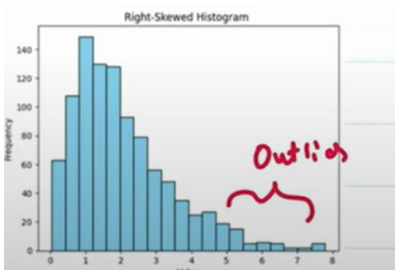


- Here most of the data points lies between 0 to 20 but few lies between 75 to 80 so we can say dataset has outliers

Basically Output / Result will be shown in 3 Way in Histogram:

- Skewness tells us about the shape of a data distribution — specifically, whether the data is symmetrical or tilted (skewed) to one side.
- If the data is evenly spread around the mean → it is not skewed (symmetrical).
- If the data has a long tail on one side, it's skewed.

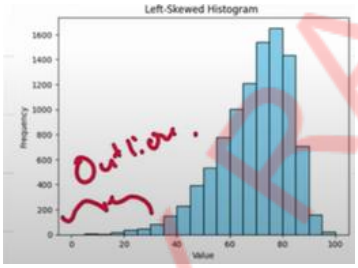
1. **Right Skewed:**



Meaning:

- Many small values, few large values.
- Long tail on the **right side** (high values).
- **Outliers likely on the high side**.

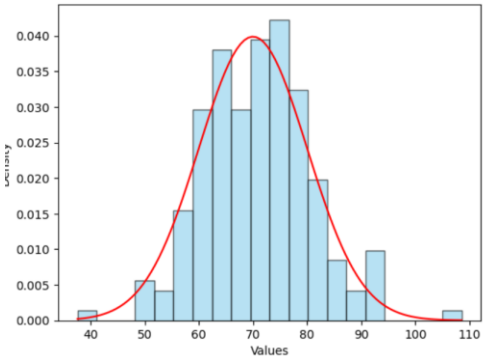
2. **Left Skewed:**



Meaning:

- Many high values, few small values.
- Long tail on the **left side** (low values).
- **Outliers likely on the low side**.

3. Normal Distribution:



Meaning:

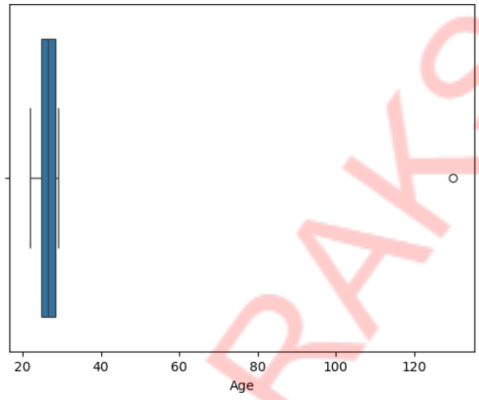
- Data is balanced around the centre (mean ≈ median).
- No significant skewness.
- Outliers are rare (if any).
- The Graph will be peak at the middle

---

4. Using Box plot

```
plt.boxplot(x = df["Age"])
```

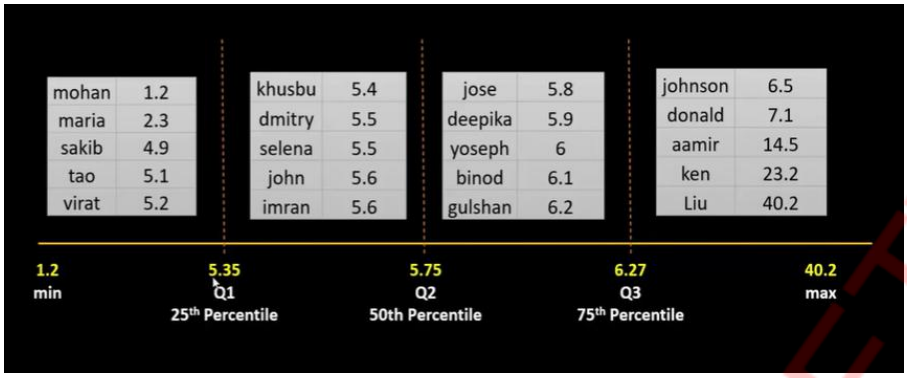
- A **boxplot** is a **visual chart** that shows:
- How your data is spread (distribution)
- Where most values lie (middle 50%)
- Where the **outliers** are
- It is based on **IQR (Interquartile Range)**.



Removing Outliers

1. Outlier Removal using IQR Method:

- IQR measures the spread of the middle 50% of the data.



- Here first we need to understand 25th percentile
- Means here 25th percentile = 5.35
- Meaning there are 25%of data which is less than 5.35
- Similarly, 50th percentile and 75th percentile

Detecting Outliers

```
import pandas as pd

df = pd.DataFrame({'Score': [50, 55, 60, 200, 65, 58, 59]})

Q1 = df['Score'].quantile(0.25)
Q3 = df['Score'].quantile(0.75)
IQR = Q3 - Q1

lower_limit = Q1 - 1.5 * IQR
upper_limit = Q3 + 1.5 * IQR

print("Lower Limit:", lower_limit)
print("Upper Limit:", upper_limit)
```

Output Example:

```
yaml

Lower Limit: 44.0
Upper Limit: 87.5
```

- Any value < 44 or > 87.5 is an **outlier**.
- In this case, `200` is an outlier.

3️⃣ Trimming (Removing Outliers)

- **Trimming** means **removing rows** with outliers.

```python
df_trimmed = df[(df['Score'] >= lower_limit) & (df['Score'] <= upper_limit)]
print(df_trimmed)
```

Output:

```nginx
   Score
0     50
1     55
2     60
4     65
5     58
6     59
```

✅ **Explanation:**

- The outlier `200` is removed.
- Dataset size **reduces**.
- Use trimming when **dataset is large** and removing a few rows **won't affect analysis**.

4️⃣ Capping / Winsorization

- **Capping** means **replacing outliers** with the **boundary value** instead of removing them.
- Useful when **dataset is small** and you don't want to lose data.

```python
import numpy as np

df['Score_capped'] = np.where(df['Score'] > upper_limit, upper_limit,
                              np.where(df['Score'] < lower_limit, lower_limit, df['Score']))
print(df)
```

Output:

```nginx
   Score  Score_capped
0     50          50.0
1     55          55.0
2     60          60.0
3    200          87.5
4     65          65.0
5     58          58.0
6     59          59.0
```

✅ **Explanation:**

- Outlier `200` is **capped** to the upper limit `87.5`.
- Dataset size **remains the same**.
- Capping reduces the **effect of extreme values** on statistics like mean or correlation.

- The number 1.5 was chosen based on statistics and probability theory.
  It comes from the idea that for a normal (bell-shaped) distribution,
  almost all (about 99.3%) of data should lie within ±1.5 × IQR from the middle 50%.

Syntax of np.where: np.where is similar to if-else statement

```
np.where(condition, value_if_true, value_if_false)
```

- **condition** → a boolean condition (e.g., `arr > 10`)
- **value_if_true** → value to assign if the condition is True
- **value_if_false** → value to assign if the condition is False

---

2 using Z-score Method

- Z-score measures how many **standard deviations** a data point is from the **mean**.
- Formula:

$$Z = \frac{X - \text{Mean}}{\text{Standard Deviation}}$$

Rule of thumb: If |Z| > 3 → considered an **outlier**.

2️⃣ Detecting Outliers Using Z-score

```python
import pandas as pd
import numpy as np
from scipy import stats

# Sample data
df = pd.DataFrame({'Score': [50, 55, 60, 200, 65, 58, 59]})

# Calculate Z-score
z = np.abs(stats.zscore(df['Score']))
print(z)
```

Output Example:

```csharp
[0.845, 0.423, 0.070, 5.090, 0.497, 0.211, 0.140]
```

- Values with **Z-score > 3** are considered outliers.
- In this case, `200` is an outlier (Z = 5.09).

.

3️⃣ Trimming (Removing Outliers)

- **Trimming** means **removing rows** with Z-score above the threshold.

```python
df_trimmed = df[z < 3]  # Keep only rows with Z < 3
print(df_trimmed)
```

Output:

```nginx
   Score
0     50
1     55
2     60
4     65
5     58
6     59
```

✅ **Explanation:**

- Outlier `200` is removed.
- Dataset size **reduces**.
- Use when dataset is **large** and removing a few rows **won't affect analysis**.

4️⃣ Capping / Winsorization

- **Capping** means **replacing outliers** with the **boundary value** (mean ± threshold × std).
- Useful for **small datasets** or when you **don't want to remove data**.

```python
mean = df['Score'].mean()
std = df['Score'].std()

upper_limit = mean + 3*std
lower_limit = mean - 3*std

df['Score_capped'] = np.where(df['Score'] > upper_limit, upper_limit,
                              np.where(df['Score'] < lower_limit, lower_limit, df['Score']))
print(df)
```

Output:

```nginx
   Score  Score_capped
0     50          50.0
1     55          55.0
2     60          60.0
3    200         107.1
4     65          65.0
5     58          58.0
6     59          59.0
```

✅ **Explanation:**

- Outlier `200` is **capped** to the **upper limit** (mean + 3×std ≈ 107.1).
- Dataset size **remains the same**.
- Reduces the **effect of extreme values** on statistics like mean, correlation, or regression.

3 Using Percentile Method

- **Percentiles** divide your data into 100 equal parts.
- Example: 1st percentile = value below which 1% of data falls, 99th percentile = value below which 99% of data falls.
- Outliers are **values outside a chosen percentile range** (commonly 1st–99th or 5th–95th percentiles).
- **Flexible:** You can adjust percentiles depending on the dataset.
- Useful for **skewed or non-normal data**.

2 Detecting Outliers Using Percentiles

```python
import pandas as pd

df = pd.DataFrame({'Score': [50, 55, 60, 200, 65, 58, 59]})

# Define percentile limits
lower_limit = df['Score'].quantile(0.01)  # 1st percentile
upper_limit = df['Score'].quantile(0.99)  # 99th percentile

print("Lower Limit:", lower_limit)
print("Upper Limit:", upper_limit)
```

Output Example:

```yaml
Lower Limit: 50.0
Upper Limit: 200.0
```

- Any value **below 1st percentile** or **above 99th percentile** is an **outlier**.

3 Trimming (Removing Outliers)

- **Trimming** removes rows outside the percentile limits.

```python
df_trimmed = df[(df['Score'] >= lower_limit) & (df['Score'] <= upper_limit)]
print(df_trimmed)
```

Output:

```nginx
   Score
0     50
1     55
2     60
3    200
4     65
5     58
6     59
```

✅ **Explanation:**

- In this example, no value falls outside 1–99 percentile, so nothing is removed.
- If you use **5th–95th percentile**, `200` would be trimmed.

4️⃣ Capping / Winsorization

- **Capping** replaces values beyond percentile limits with **percentile boundaries**.

```python
import numpy as np

df['Score_capped'] = np.where(df['Score'] > upper_limit, upper_limit,
                              np.where(df['Score'] < lower_limit, lower_limit, df['Score']))
print(df)
```

Output:

```
   Score  Score_capped
0     50          50.0
1     55          55.0
2     60          60.0
3    200         200.0
4     65          65.0
5     58          58.0
6     59          59.0
```

✅ **Explanation:**

- Values above 99th percentile replaced with 99th percentile.
- Values below 1st percentile replaced with 1st percentile.
- Dataset size **remains the same**.

If an interviewer asks “how do you handle outliers?”

“I usually use IQR/Quantile for skewed or general data, Z-score for normally distributed data, and Percentiles for flexible trimming/capping. Depending on the dataset, I either remove extreme values (trimming) or replace them with boundaries (capping). These three methods are enough for most scenarios.”

Filtering Rows

1. Basic Boolean Filtering (Single Condition)

- You **filter rows** based on a condition using:

```
df[df['column_name'] == some_value]
```

```
df[df['Age'] > 25]
```

This returns only the rows where the `Age` column has values greater than 25.

2. Filtering with Multiple Conditions (AND / OR)

- AND condition: Use & and wrap conditions in brackets:

```
df[(df['Age'] > 25) & (df['Gender'] == 'Male')]
```

- **OR condition**: Use |:

```
df[(df['Age'] < 18) | (df['Age'] > 60)]
```

Important: Always put each condition inside **brackets ( )** — otherwise you'll get an error.

3. Filtering using isin() – For Multiple Values

- Use this when you want to match multiple values from a list:

```
df[df['Department'].isin(['HR', 'Finance'])]
```

This returns all rows where the `Department` is either `'HR'` or `'Finance'`.

4. Filtering using ~ (NOT condition)

- Use ~ to exclude rows that match a condition.

```
df[~df['Department'].isin(['HR', 'Finance'])]
```

This returns rows **not** in `'HR'` or `'Finance'`.

5. Filtering using .str methods (for text columns)

- Use these to filter text values (strings):

```
df[df['Name'].str.startswith('A')]
df[df['Email'].str.contains('@gmail.com')]
df[df['City'].str.lower() == 'bangalore']  # case insensitive
```

> Note: Use `.str.lower()` for **case-insensitive** comparisons.

6. Filtering using .between() for ranges

- This is cleaner than using >= and <=:

```
df[df['Age'].between(25, 35)]
```

This gives rows where Age is between 25 and 35 (inclusive).

OR

```
df[ (df['Age'] >= 25 ) & (df['Age'] <= 35)]
```

---

7. Filtering using .query () method (optional but useful)

- Let's you write conditions like SQL:

```
df.query("Age > 30 and Gender == 'Female'")
```

✅ Easy to read

⚠️ Strings must be quoted (`'Female'` in quotes)

Edge Cases:

---

1. Missing values (NaN)

- If a column has NaN (missing values), direct comparisons like df[df['Age'] > 25] will exclude those rows silently.
- To handle missing values:

```
df[df["Age"].isna()]  # Keeps rows where age is null
```

```
df[df["Age"].notna()]  # Keeps rows where age is not null
```

---

2. Using == with strings that have spaces or inconsistent casing

- Make sure to clean the data first:

```
df['City'] = df['City'].str.strip().str.lower()
df[df['City'] == 'mumbai']
```

---

3. Don't forget parentheses with multiple conditions

- This will throw an error:

```
python
df[df['Age'] > 25 & df['Gender'] == 'Male']
```

❌

Instead, do:

```
python
df[(df['Age'] > 25) & (df['Gender'] == 'Male')]
```

✅

---

4. .str methods only work on string columns

- If the column has mixed types or nulls, convert first:

If the column has mixed types or nulls, convert first:

```python
df['Name'] = df['Name'].astype(str)
```

---

5. Case sensitivity

- Most string operations are case-sensitive by default. Use .str.lower() or .str.upper() to normalize.

- **Column Doesn't Exist**

- df['salary'] ❌ if it's actually 'Salary'
- Pandas is **case-sensitive**. Always check exact spellings and cases.

Sort Values

- sort_values () in Pandas is used to arrange or reorder rows in a DataFrame or Series based on the values in one or more columns.
- It helps you organize data in ascending or descending order — which makes analysis, comparison, and visualization much easier.

---

1. Basic Sorting by a Single Column

```
df.sort_values('column_name')
```

- By default, this sorts the column in ascending order.

Example:

```
df.sort_values('Age')
```

This gives you the data sorted from youngest to oldest.

---

2. Descending Order (Most Common Use Case)

Use ascending=False to sort from high to low:

```
df.sort_values('Salary', ascending=False)
```

This helps you find top values (e.g., highest salary, top scores).

---

3. Sorting by Multiple Columns

Use a **list of columns** to sort by priority:

```
df.sort_values(['Department', 'Age'])
```

- This sorts by Department first (A–Z),
- Then within each department, it sorts by Age (low to high).

Add ascending = [True, False] for mixed directions:

```
df.sort_values(['Department', 'Age'], ascending=[True, False])
```

---

4. Sorting by Index

Sometimes you may want to sort rows based on their **index (row labels)** instead of column values:

```
df.sort_index()
```

Use `ascending=False` for reverse index order.

---

5. In-Place Sorting

By default, Pandas returns a **new Data Frame**.

Use inplace =True to **modify the original Data Frame**:

```
df.sort_values('Age', inplace=True)
```

⚠️ Be careful — this **doesn't return anything**, so don't assign it like `df = df.sort_values(...)` if `inplace=True`.

Agg () function in Pandas

We use the **agg () function** when we want to apply **aggregation functions (like sum, mean, min, max, count)** on either:

1. **All columns at once**:
   Instead of writing separate lines to calculate sum or mean for every column, we apply them in one go.
2. **Specific set of columns with specific functions**:
   We can customize the aggregation for each column, just like how in Excel you might use different formulas for different columns in a summary table.

1. ✅ Applying multiple functions to all columns (r1)

```python
df.agg(['sum', 'min', 'max'])
```

→ Very common in data summarization tasks.

Must know.

2. ✅ Applying different functions to specific columns (r2)

```python
df.agg({
    'A': ['sum', 'min', 'max'],
    'C': ['sum', 'min', 'max']
})
```

→ Frequently used when you want tailored aggregation per column.

Must know.

1️⃣ Using agg() on a Series (single column)

👉 When you want to summarize one column.

```
import pandas as pd
import numpy as np

df = pd.DataFrame({
    'Sales': [200, 300, 400, 500, np.nan]
})
```

```
df
```

	Sales
0	200.0
1	300.0
2	400.0
3	500.0
4	NaN

```
# Using one function
print(df['Sales'].agg('mean'))
```

```
350.0
```

```
# Using multiple functions
print(df['Sales'].agg(['min', 'max', 'mean', 'count']))
```

```
min      200.0
max      500.0
mean     350.0
count      4.0
Name: Sales, dtype: float64
```

2️⃣ Using agg() on a DataFrame (multiple columns)

👉 When you want to summarize many columns together.

```
df = pd.DataFrame({
    'Sales': [200, 300, 400, 500],
    'Profit': [20, 30, 40, 50]
})
```

```
df
```

	Sales	Profit
0	200	20
1	300	30
2	400	40
3	500	50

```
# Apply one function to all columns
print(df.agg('mean'))
```

```
Sales     350.0
Profit     35.0
dtype: float64
```

```
df.agg({'Sales': ['sum', 'mean'], 'Profit': 'max'})
```

	Sales	Profit
sum	1400.0	NaN
mean	350.0	NaN
max	NaN	50.0

	Employee_ID	Name	Department	Salary	Experience_Years
3	104	Sneha	IT	80000	6
2	103	Arjun	Finance	72000	5
5	106	Meena	Finance	72000	7
0	101	Rahul	IT	65000	3
7	108	Anjali	IT	60000	3
4	105	Kiran	Marketing	60000	4
1	102	Priya	HR	55000	2
6	107	Rajesh	HR	50000	1

```
df['Salary'].agg('sum')
```

```
514000
```

Example 3:

```
df.agg(["sum","min","max"])
```

	Employee_ID	Name	Department	Salary	Experience_Years
sum	836	SnehaArjunMeenaRahulAnjaliKiranPriyaRajesh	ITFinanceFinanceITITMarketingHRHR	514000	31
min	101	Anjali	Finance	50000	1
max	108	Sneha	Marketing	80000	7

Example 4:

```
df.agg({"Salary":["min","max","sum"],
        "Experience_Years":["min","max","sum"]})
```

	Salary	Experience_Years
min	50000	1
max	80000	7
sum	514000	31

Example 5:

```
df.agg("min",axis=0)
```

```
Employee_ID              101
Name                   Anjali
Department            Finance
Salary                  50000
Experience_Years            1
dtype: object
```

Example 7:

```
df.agg("sum",axis=0)
```

```
Employee_ID                                                  836
Name                 SnehaArjunMeenaRahulAnjaliKiranPriyaRajesh
Department                     ITFinanceFinanceITITMarketingHRHR
Salary                                                   514000
Experience_Years                                             31
dtype: object
```

Example 8:

```
df.agg({'Salary': ['min', 'max'], 'Experience_Years': 'mean'})
```

	Salary	Experience_Years
min	50000.0	NaN
max	80000.0	NaN
mean	NaN	3.875

```
df[["Salary","Experience","Bonus"]].agg(["sum","min","max"])
```

	Salary	Experience	Bonus
sum	407000	32	21900
min	40000	2	2000
max	60000	7	4000

Group By in Pandas

groupby is a **powerful function in Pandas** that allows you to **split your data into groups based on one or more columns**, perform an **aggregation or transformation** on each group, and then **combine the results**.

Think of it as **"split → apply → combine"**:

- **Split**: Divide the data into groups.
- **Apply**: Perform some operation (sum, mean, count, etc.) on each group.
- **Combine**: Return a new DataFrame or Series with the results.
- Similar to SQL groupby

Syntax:

```
df.groupby('column_name')['another_column'].operation()
```

- `column_name` : the column to group by
- `another_column` : the column you want to apply the function on
- `operation` : like `sum()` , `mean()` , `count()` , `max()` , etc.

Example:

```
import pandas as pd

# Sample data
data = {
    'Department': ['IT', 'HR', 'Finance', 'IT', 'HR', 'Finance', 'IT'],
    'Employee': ['Alice', 'Bob', 'Charlie', 'David', 'Eva', 'Frank', 'Grace'],
    'Salary': [60000, 50000, 55000, 65000, 52000, 58000, 70000]
}

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

	Department	Employee	Salary
0	IT	Alice	60000
1	HR	Bob	50000
2	Finance	Charlie	55000
3	IT	David	65000
4	HR	Eva	52000
5	Finance	Frank	58000
6	IT	Grace	70000

```
# Group by department and calculate mean salary
df.groupby('Department')['Salary'].mean()
```

```
Department
Finance    56500.0
HR         51000.0
IT         65000.0
Name: Salary, dtype: float64
```

```
# Group by department and calculate count salary
df.groupby('Department')['Salary'].count()
```

```
Department
Finance    2
HR         2
IT         3
Name: Salary, dtype: int64
```

```
# Group by department and calculate mean salary
df.groupby('Department')['Salary'].agg(["min","max","sum"])
```

Department	min	max	sum
Finance	55000	58000	113000
HR	50000	52000	102000
IT	60000	70000	195000

Group By Multiple Columns ¶

```
data2 = {
    'Department': ['IT', 'IT', 'HR', 'HR'],
    'Gender': ['M', 'F', 'M', 'F'],
    'Salary': [60000, 65000, 50000, 52000]
}

df2 = pd.DataFrame(data2)

df2.groupby(['Department', 'Gender'])['Salary'].mean()
```

```
Department  Gender
HR          F         52000.0
            M         50000.0
IT          F         65000.0
            M         60000.0
Name: Salary, dtype: float64
```

[52]:
```
df.groupby('Department').agg({
    'Salary': ['mean', 'sum', 'max'],
    'Employee': 'count'   # count employees per department
})
```

[52]:

Department	Salary			Employee
	mean	sum	max	count
Finance	56500.0	113000	58000	2
HR	51000.0	102000	52000	2
IT	65000.0	195000	70000	3

Merging Data Frames

- merge is a function in Pandas used to combine two Data Frames based on common columns or indices, similar to SQL joins.
- Library: pandas
- Function: pd.merge()
- It helps you combine datasets efficiently for analysis.

Basic Syntax:

```
import pandas as pd

merged_df = pd.merge(left_df, right_df, how='inner', on='key_column')
```

Parameters explained:

Parameter	Description
`left`	The first DataFrame.
`right`	The second DataFrame.
`how`	Type of merge: `'inner'`, `'left'`, `'right'`, `'outer'` (explained below).
`on`	Column name(s) to join on. Both DataFrames must have this column.
`left_on`	Column(s) from left DataFrame if names differ.
`right_on`	Column(s) from right DataFrame if names differ.
`suffixes`	Tuple to append to overlapping column names (default `('_x','_y')`).

3. Types of Merge (`how` parameter)

Think of it like SQL joins:

1. **Inner Join** (default)
   - Keeps **only rows that match** in both DataFrames.
   - Example:

```
python

pd.merge(df1, df2, on='id', how='inner')
```

- Only IDs present in **both df1 and df2** will appear.

2. **Left Join**
   - Keeps **all rows from the left DataFrame**, matching rows from right.
   - Missing values from right → `NaN`.

```
python

pd.merge(df1, df2, on='id', how='left')
```

3. **Right Join**
   - Keeps **all rows from the right DataFrame**, matching rows from left.
   - Missing values from left → `NaN`.
4. **Outer Join**
   - Keeps **all rows from both DataFrames**.
   - Non-matching rows → `NaN`.

```python
pd.merge(df1, df2, on='id', how='outer')
```

4. Merge on Multiple Columns

If two columns together form a key:

```python
pd.merge(df1, df2, on=['id', 'date'], how='inner')
```

5. Merge with Different Column Names

If the key column names are different in left and right DataFrames:

```python
pd.merge(df1, df2, left_on='id_left', right_on='id_right', how='inner')
```

The 4 Most Common Merge Types (the 80% you'll use)

6. Handling Overlapping Column Names

If both DataFrames have **other columns with the same name**, Pandas automatically adds suffixes `_x` and `_y`. You can change them:

```python
pd.merge(df1, df2, on='id', suffixes=('_left', '_right'))
```

Copy code

Examples:

`df1`

	emp_id	date	sales	region
0	1	2025-01-01	200	East
1	2	2025-01-02	300	West
2	3	2025-01-03	400	East
3	4	2025-01-04	500	North

`df2`

	id_emp	date	name	region
0	2	2025-01-02	Alice	West
1	3	2025-01-03	Bob	East
2	4	2025-01-04	Charlie	North
3	5	2025-01-05	David	South

```python
inner_merge = pd.merge(df1, df2, left_on='emp_id', right_on='id_emp', how='inner')
print(inner_merge)
```

```
   emp_id      date_x  sales region_x  id_emp      date_y     name region_y
0       2  2025-01-02    300     West       2  2025-01-02    Alice     West
1       3  2025-01-03    400     East       3  2025-01-03      Bob     East
2       4  2025-01-04    500    North       4  2025-01-04  Charlie    North
```

```
left_merge = pd.merge(df1, df2, left_on='emp_id', right_on='id_emp', how='left')
print(left_merge)
```

```
   emp_id      date_x  sales region_x  id_emp      date_y     name region_y
0       1  2025-01-01    200     East     NaN         NaN      NaN      NaN
1       2  2025-01-02    300     West     2.0  2025-01-02    Alice     West
2       3  2025-01-03    400     East     3.0  2025-01-03      Bob     East
3       4  2025-01-04    500    North     4.0  2025-01-04  Charlie    North
```

```
right_merge = pd.merge(df1, df2, left_on='emp_id', right_on='id_emp', how='right')
print(right_merge)
```

```
   emp_id      date_x  sales region_x  id_emp      date_y     name region_y
0     2.0  2025-01-02  300.0     West       2  2025-01-02    Alice     West
1     3.0  2025-01-03  400.0     East       3  2025-01-03      Bob     East
2     4.0  2025-01-04  500.0    North       4  2025-01-04  Charlie    North
3     NaN         NaN    NaN      NaN       5  2025-01-05    David    South
```

```
outer_merge = pd.merge(df1, df2, left_on='emp_id', right_on='id_emp', how='outer')
print(outer_merge)
```

```
   emp_id      date_x  sales region_x  id_emp      date_y     name region_y
0     1.0  2025-01-01  200.0     East     NaN         NaN      NaN      NaN
1     2.0  2025-01-02  300.0     West     2.0  2025-01-02    Alice     West
2     3.0  2025-01-03  400.0     East     3.0  2025-01-03      Bob     East
3     4.0  2025-01-04  500.0    North     4.0  2025-01-04  Charlie    North
4     NaN         NaN    NaN      NaN     5.0  2025-01-05    David    South
```

```
multi_merge = pd.merge(df1, df2, left_on=['emp_id','date'], right_on=['id_emp','date'], how='inner')
print(multi_merge)
```

```
   emp_id        date  sales region_x  id_emp     name region_y
0       2  2025-01-02    300     West       2    Alice     West
1       3  2025-01-03    400     East       3      Bob     East
2       4  2025-01-04    500    North       4  Charlie    North
```

```
overlap_merge = pd.merge(df1, df2, left_on='emp_id', right_on='id_emp', how='inner', suffixes=('_sales','_info'))
overlap_merge
```

	emp_id	date_sales	sales	region_sales	id_emp	date_info	name	region_info
0	2	2025-01-02	300	West	2	2025-01-02	Alice	West
1	3	2025-01-03	400	East	3	2025-01-03	Bob	East
2	4	2025-01-04	500	North	4	2025-01-04	Charlie	North

```
df1
```

	sales	region
A	200	East
B	300	West
C	400	North

```
df2
```

	name	dept
B	Alice	HR
C	Bob	IT
D	Charlie	Finance

```python
# Merge based on index
merged_index = pd.merge(df1, df2, left_index=True, right_index=True, how='inner')
print(merged_index)
```

```
   sales region   name dept
B    300   West  Alice   HR
C    400  North    Bob   IT
```

```python
# Merge based on index
merged_index = pd.merge(df1, df2, left_index=True, right_index=True, how='left')
print(merged_index)
```

```
   sales region   name dept
A    200   East    NaN  NaN
B    300   West  Alice   HR
C    400  North    Bob   IT
```

```python
# Merge based on index
merged_index = pd.merge(df1, df2, left_index=True, right_index=True, how='right')
print(merged_index)
```

```python
# Merge based on index
merged_index = pd.merge(df1, df2, left_index=True, right_index=True, how='outer')
print(merged_index)
```

```
   sales region     name     dept
A  200.0   East      NaN      NaN
B  300.0   West    Alice       HR
C  400.0  North      Bob       IT
D    NaN    NaN  Charlie  Finance
```

To track from which data frame row came

```python
result_indicator = pd.merge(df1, df2, on='dept_id', how='outer', indicator=True)
print(result_indicator)
```

```
   emp_id     name  dept_id dept_name      _merge
0     1.0    Alice      101        HR        both
1     2.0      Bob      102   Finance        both
2     3.0  Charlie      103       NaN   left_only
3     4.0    David      104       NaN   left_only
4     NaN      NaN      105        IT  right_only
```

7. Common Edge Cases & Things to Watch

1. **No matching column**
   - If the `on` column doesn't exist in both DataFrames → **KeyError**.
2. **Duplicate keys**
   - If your key column has **duplicate values**, merge will create a **Cartesian product** of matches.

python Copy code

```
# Example: df1 has 2 rows with id=1, df2 has 3 rows with id=1 → merged = 6 rows
```

3. **Different data types in key columns**
    - Merge fails silently or gives 0 matches if types differ (e.g., `int` vs `str`).
    - Fix: `df['id'] = df['id'].astype(int)`
4. **Missing values in key columns**
    - By default, `NaN` keys **do not match**, even in outer join.
5. **Large datasets**
    - Merge can be slow with very large DataFrames; using `merge` on **indexed columns** or pre-filtering can help.

Concat () Function

- pandas.concat() is used to **combine two or more pandas objects (DataFrames or Series) along a particular axis**.
- **Axis 0 → Vertical (rows)** → Stack DataFrames one below the other
- **Axis 1 → Horizontal (columns)** → Join DataFrames side by side

Think of it like:

- **Axis 0:** Adding more records
- **Axis 1:** Adding more columns

Basic Syntax:

```
import pandas as pd

pd.concat(objs, axis=0, join='outer', ignore_index=False, keys=None)
```

Parameters:

Parameter	What it does
`objs`	List of DataFrames or Series to concatenate
`axis`	0 → rows, 1 → columns
`join`	`outer` (union of columns), `inner` (intersection of columns)
`ignore_index`	True → reset the index in the result, False → keep original indexes
`keys`	Assigns a hierarchical index (multiindex) to each concatenated object

Vertical Concatenation (Axis = 0)

```
import pandas as pd

df1 = pd.DataFrame({'Name': ['Alice', 'Bob'], 'Age': [25, 30]})
df2 = pd.DataFrame({'Name': ['Charlie', 'David'], 'Age': [35, 40]})
df1
```

	Name	Age
0	Alice	25
1	Bob	30

```
df2
```

	Name	Age
0	Charlie	35
1	David	40

```
result = pd.concat([df1, df2], axis=0)
print(result)
```

```
      Name  Age
0    Alice   25
1      Bob   30
0  Charlie   35
1    David   40
```

- ✅ Notice: The index is **repeated** because ignore_index=False by default.
- **With ignore_index=True:**

```
result = pd.concat([df1, df2], axis=0, ignore_index=True)
result
```

	Name	Age
0	Alice	25
1	Bob	30
2	Charlie	35
3	David	40

Horizontal Concatenation (Axis = 1)

```
df1 = pd.DataFrame({'Name': ['Alice', 'Bob']})
df2 = pd.DataFrame({'Age': [25, 30]})

result = pd.concat([df1, df2], axis=1)
print(result)
```

```
    Name  Age
0  Alice   25
1    Bob   30
```

Handling Different Columns (join parameter)

Outer Join (default):

```
df1 = pd.DataFrame({'A': [1,2]})
df2 = pd.DataFrame({'B': [3,4]})

result = pd.concat([df1, df2], axis=0, join='outer')
result
```

	A	B
0	1.0	NaN
1	2.0	NaN
0	NaN	3.0
1	NaN	4.0

```
# Inner Join
result = pd.concat([df1, df2], axis=0, join='inner')
result
```

	A	B
0	1	3
1	2	4

- **Key point:** inner keeps only **common columns**, outer keeps **all columns**.

Using keys to create a hierarchical index

```
result = pd.concat([df1, df2], keys=['Group1','Group2'])
print(result)
```

```
            A    B
Group1 0  1.0  NaN
       1  2.0  NaN
Group2 0  NaN  3.0
       1  NaN  4.0
```

Common Edge Cases

1. Different column names: NaNs will appear if columns are missing in some DataFrames.
2. Index duplication: By default, indexes are not reset, so duplicate indexes can appear. Use ignore_index=True to avoid confusion.
3. Data type mismatch: Concatenating columns with different types may upcast to a common type (e.g., int → float).
4. Axis confusion: Axis 0 = vertical (rows), Axis 1 = horizontal (columns). Many beginners mix them up.
5. Merging vs concatenating: concat does not align on keys like SQL joins; it's mostly stacking. For key-based merges, use pd.merge().

---

join ()

- Join () in pandas is used to **combine two DataFrames based on their indexes** (by default) or on a key column (if specified).
- **Think of it as:** "Combine two tables side by side — match rows using their index or a key column."

Basic Syntax:

```
DataFrame.join(other, on=None, how='left', lsuffix='', rsuffix='', sort=False)
```

Parameter	Meaning
`other`	The DataFrame to join with
`on`	Column name to join on (if not using index)
`how`	Type of join: `'left'`, `'right'`, `'outer'`, `'inner'`
`lsuffix` / `rsuffix`	Add suffixes if column names overlap
`sort`	Sort the result DataFrame by the join keys

Type	Description	SQL Equivalent
`left`	Keep all rows from the left DataFrame	LEFT JOIN
`right`	Keep all rows from the right DataFrame	RIGHT JOIN
`inner`	Keep only rows with matching keys in both DataFrames	INNER JOIN
`outer`	Keep all rows from both DataFrames	FULL OUTER JOIN

```
import pandas as pd

df1 = pd.DataFrame({
    'emp_id': [1, 2, 3],
    'name': ['Alice', 'Bob', 'Charlie']
}).set_index('emp_id')

df2 = pd.DataFrame({
    'emp_id': [2, 3, 4],
    'salary': [50000, 60000, 70000]
}).set_index('emp_id')
```

emp_id	name
1	Alice
2	Bob
3	Charlie

emp_id	salary
2	50000
3	60000
4	70000

```
# Left Join (Default)

result = df1.join(df2, how='left')
result
```

emp_id	name	salary
1	Alice	NaN
2	Bob	50000.0
3	Charlie	60000.0

- Keeps **all records from the left DataFrame (df1)**.
- Missing matches in df2 become **NaN**.
- Most commonly used type of join for data analysis.

```
# Right Join
result = df1.join(df2, how='right')
result
```

emp_id	name	salary
2	Bob	50000
3	Charlie	60000
4	NaN	70000

- Keeps **all records from the right DataFrame (df2)**.
- Missing values from df1 become **NaN**.

```
# Inner Join
result = df1.join(df2, how='inner')
result
```

emp_id	name	salary
2	Bob	50000
3	Charlie	60000

- Keeps **only matching keys** from both DataFrames.
- Most used when you only need **common data**.

```
# Outer Join
result = df1.join(df2, how='outer')
result
```

emp_id	name	salary
1	Alice	NaN
2	Bob	50000.0
3	Charlie	60000.0
4	NaN	70000.0

```python
# Join using a column (Not a Index)

df1 = pd.DataFrame({'emp_id': [1, 2, 3], 'name': ['Alice', 'Bob', 'Charlie']})
df2 = pd.DataFrame({'emp_id': [2, 3, 4], 'salary': [50000, 60000, 70000]})

result = df1.join(df2.set_index('emp_id'), on='emp_id', how='left')
result
```

	emp_id	name	salary
0	1	Alice	NaN
1	2	Bob	50000.0
2	3	Charlie	60000.0

```python
# Handling Duplicate Column Names (lsuffix & rsuffix)
# If both DataFrames have a column with the same name, you'll get an error.

df1 = pd.DataFrame({'A': [1, 2], 'B': [3, 4]})
df2 = pd.DataFrame({'A': [5, 6], 'C': [7, 8]})

result = df1.join(df2, lsuffix='_left', rsuffix='_right')
result
```

	A_left	B	A_right	C
0	1	3	5	7
1	2	4	6	8

Common Edge Cases

1. Index mismatch

- If indexes don't align, you'll get NaNs or misaligned data.
- Tip: Use set_index() before join or reset indexes carefully.

2. Overlapping column names

- You'll get a ValueError unless you use lsuffix and rsuffix.

3. Data type mismatch in keys

- Example: one key is int, the other is str. → No match found.
- Always ensure key columns have the same dtype.

4. Duplicate index values

- If either DataFrame has duplicate index values, join results in duplicate rows.

5. Joining on different column names

- join() can only join on index or one column (if specified).
- For complex joins (different column names or multiple keys), use pd.merge() instead.

Pivot

- A pivot is used to reshape your Data Frame — it helps you summarize data or change rows into columns for better analysis.
- It's similar to an Excel Pivot Table, but in code form.
- In simple words: Pivot helps you reorganize data — turning unique values from one column into new columns.

When to Use pivot ()

- Use pivot () when:
- Your data is in **long format** (rows represent observations)
- You want to convert it to **wide format**
- Each combination of index and columns is **unique**

Syntax:

```
df.pivot(index='row_column', columns='column_to_pivot', values='value_column')
```

Parameters:

- `index` → The column to use as the new **rows**
- `columns` → The column to use as the new **columns**
- `values` → The column whose values fill the table

- Here values optional

Examples:

	Date	Product	Units	Revenue	Others
0	2025-10-01	Laptop	5	50000	A
1	2025-10-01	Phone	10	20000	B
2	2025-10-02	Laptop	4	40000	C
3	2025-10-02	Phone	8	16000	D

Example 1:

```
df.pivot(index='Date',columns='Product')
```

	Units		Revenue		Others	
Product	**Laptop**	**Phone**	**Laptop**	**Phone**	**Laptop**	**Phone**
Date						
2025-10-01	5	10	50000	20000	A	B
2025-10-02	4	8	40000	16000	C	D

- If we not give values all remaining columns treated as values other than index and columns.
- Even if Remaining column is Categorical and numerical also.

Example 2:

- If we give values only those columns will be selected not all.

```
df.pivot(index='Date',columns='Product',values='Revenue')
```

Product	Laptop	Phone
Date		
2025-10-01	50000	20000
2025-10-02	40000	16000

Example 3:

- We can give More than one values as well

```
df.pivot(index='Date',columns='Product',values=['Revenue','Units'])
```

	Revenue		Units	
Product	Laptop	Phone	Laptop	Phone
Date				
2025-10-01	50000	20000	5	10
2025-10-02	40000	16000	4	8

Pivot_table

- Pivot table is used to summarize and aggregate data inside a dataframe.
- Both pivot () and pivot_table () are used to reshape data (convert long → wide format), but only pivot_table () can aggregate data.
  - **Pivot ()** just *rearranges* data — it does not perform aggregation and works only if every combination of index and columns is unique.
  - **pivot_table ()** can handle both unique and duplicate combinations because it can aggregate (e.g. sum, mean, count, etc.) using the aggfunc parameter.

Syntax:

```
df.pivot_table(index='Column1', columns='Column2', values='Column3', aggfunc='sum')
```

- In aggfunc sum, max, min, count can be written.

Examples:

```
         Date Product  Sales
0  2025-10-01   Apple    100
1  2025-10-01   Apple    150
2  2025-10-02  Banana    200
3  2025-10-02  Banana    250
4  2025-10-03   Apple    300
5  2025-10-03  Banana    100
```

Example 1:

```
df1.pivot_table(index='Date',columns='Product')
```

	Sales	
Product	Apple	Banana
Date		
2025-10-01	125.0	NaN
2025-10-02	NaN	225.0
2025-10-03	300.0	100.0

```
df1.pivot_table(index='Date',columns='Product')
```

	Sales		Sales1	
Product	Apple	Banana	Apple	Banana
Date				
2025-10-01	125.0	NaN	250.0	NaN
2025-10-02	NaN	225.0	NaN	450.0
2025-10-03	300.0	100.0	600.0	700.0

- In pivot_table if we not specify values and other than index and columns only numeric column is there it will not show error otherwise it will show error.
- Other than index and columns if it contains any categorical column it will show error.

Example 2:

- In pivot_table if we will not give aggfunc by default it will take it as Mean/ Average.
- And also if there is duplicate index, column combination is there it will not give error but pivot function will give.

```
df1.pivot_table(index='Date',columns='Product',values='Sales')
```

Product	Apple	Banana
Date		
2025-10-01	125.0	NaN
2025-10-02	NaN	225.0
2025-10-03	300.0	100.0

Example 3:

```
df1.pivot_table(index='Date', columns='Product', values=['Sales','Sales1'])
```

	Sales		Sales1	
Product	Apple	Banana	Apple	Banana
Date				
2025-10-01	125.0	NaN	250.0	NaN
2025-10-02	NaN	225.0	NaN	450.0
2025-10-03	300.0	100.0	600.0	700.0

Example 4:

```
df1.pivot_table(index='Date',columns='Product',values='Sales',aggfunc = 'count')
```

Product	Apple	Banana
Date		
2025-10-01	2.0	NaN
2025-10-02	NaN	2.0
2025-10-03	1.0	1.0

- Instead of count we can also give count, sum, avg, min, max, median, diff (for minus) etc.

Example 5:

```
df1.pivot_table(index='Date',columns='Product',values=['Sales','Sales1'],aggfunc = ['mean','sum'])
```

	mean				sum			
	Sales		Sales1		Sales		Sales1	
Product	Apple	Banana	Apple	Banana	Apple	Banana	Apple	Banana
Date								
2025-10-01	125.0	NaN	250.0	NaN	250.0	NaN	500.0	NaN
2025-10-02	NaN	225.0	NaN	450.0	NaN	450.0	NaN	900.0
2025-10-03	300.0	100.0	600.0	700.0	300.0	100.0	600.0	700.0

- We can give different aggregate function each will be applied for both values.

Example 6: To Apply Separate Aggregate Function to each column

```
pd.pivot_table(sales_data, values=['Units_Sold', 'Revenue'], index='Region', aggfunc={'Units_Sold': 'sum', 'Revenue': 'mean'})
```

	Revenue	Units_Sold
Region		
East	40000.000000	14
North	55000.000000	13
South	65000.000000	14
West	61666.666667	20

Example 7:

```
df1.pivot_table(index='Date',columns='Product',values=['Sales','Sales1'],aggfunc = 'sum')
```

	Sales		Sales1	
Product	Apple	Banana	Apple	Banana
Date				
2025-10-01	250.0	NaN	500.0	NaN
2025-10-02	NaN	450.0	NaN	900.0
2025-10-03	300.0	100.0	600.0	700.0

Example 8: margins = True

- When you set margins=True in a pivot_table, pandas adds an extra row and column called "All".
- These represent the totals (or overall aggregation) across all rows and columns.
- It basically gives grand totals, so you don't have to calculate them separately.

```
df1.pivot_table(index='Date',columns='Product',values=['Sales','Sales1'],aggfunc = 'count',margins=True)
```

	Sales			Sales1		
Product	Apple	Banana	All	Apple	Banana	All
Date						
2025-10-01	2.0	NaN	2	2.0	NaN	2
2025-10-02	NaN	2.0	2	NaN	2.0	2
2025-10-03	1.0	1.0	2	1.0	1.0	2
All	3.0	3.0	6	3.0	3.0	6

One Example:

	Region	Salesperson	Product	Sales	Quantity	Month
0	North	Asha	Laptop	80000	5	Jan
1	South	Vikram	Laptop	72000	4	Jan
2	East	Neha	Laptop	65000	3	Jan
3	West	Ravi	Laptop	90000	6	Jan
4	North	Asha	Mobile	50000	7	Feb
5	South	Vikram	Mobile	45000	5	Feb
6	East	Neha	Tablet	30000	2	Feb
7	West	Ravi	Tablet	40000	3	Feb
8	North	Kiran	Mobile	55000	4	Mar
9	East	Neha	Mobile	60000	5	Mar

Create a pivot table showing **total sales per product**, and then calculate the % **contribution** of each product to the overall sales.

```
pivot = df.pivot_table(index="Product",values="Sales",aggfunc="sum",margins=True)
pivot
```

Product	Sales
Laptop	307000
Mobile	210000
Tablet	70000
All	587000

```
pivot["% of Total"] = (pivot["Sales"] / pivot.loc["All", "Sales"]) * 100
pivot
```

Product	Sales	% of Total
Laptop	307000	52.299830
Mobile	210000	35.775128
Tablet	70000	11.925043
All	587000	100.000000

- Here Pivot.loc[“All”,”Sales”]

Melt Function in Pandas (Similar to Unpivot in Power Query)

- Melt () is used to convert a wide table into a long table.
- **In simple words:** It takes multiple columns and converts them into 2 columns:

Why Melt Function is Needed?

Problem with wide format

Name	Jan	Feb	Mar
A	10	20	30
B	5	15	25

If you want to:

- Calculate average sales per month
- Visualize trend month-wise
- Create Power BI charts

You need **Month & Sales columns**, not 3 separate columns.

`melt()` **converts it into:**

Name	Month	Sales
A	Jan	10
A	Feb	20
A	Mar	30
B	Jan	5
B	Feb	15
B	Mar	25

This is the **long format**, easier for:

- GroupBy
- Aggregations
- Visualization
- Machine Learning

These 4 things are the MOST important:

1 id_vars → Columns to keep as they are

These columns will **NOT melt**.

Example:

```python
pd.melt(df, id_vars=['Name'])
```

2 value_vars → Columns that will be melted

These are the columns that turn into rows.

Example:

```python
pd.melt(df, id_vars=['Name'], value_vars=['Jan', 'Feb', 'Mar'])
```

3 var_name → Rename the new column that stores the old column names

Default name = `variable`

Example:

python Copy code

```
pd.melt(df, id_vars=['Name'], var_name='Month')
```

4 value_name → Rename the column that stores the values

Default name = `value`

Example:

python

```
pd.melt(df, id_vars=['Name'], value_name='Sales')
```

Table:

	Name	Jan	Feb	Mar
0	A	10	20	30
1	B	5	15	25

Example 1:

```
df_melted = pd.melt(
    df,
    id_vars=['Name'],
    value_vars=['Jan','Feb','Mar'],
    var_name='Month',
    value_name='Sales'
)
df_melted
```

	Name	Month	Sales
0	A	Jan	10
1	B	Jan	5
2	A	Feb	20
3	B	Feb	15
4	A	Mar	30
5	B	Mar	25

Example 2: Melt Everything Except id_vars

```
pd.melt(df, id_vars='Name')
```

	Name	variable	value
0	A	Jan	10
1	B	Jan	5
2	A	Feb	20
3	B	Feb	15
4	A	Mar	30
5	B	Mar	25

- You don't need to specify value_vars always.

- This will melt all remaining columns.

```
  Region Name  Jan  Feb  Mar
0  South    A   10   20   30
1  North    B    5   15   25
```

```
pd.melt(df, id_vars=['Region', 'Name'])
```

	Region	Name	variable	value
0	South	A	Jan	10
1	North	B	Jan	5
2	South	A	Feb	20
3	North	B	Feb	15
4	South	A	Mar	30
5	North	B	Mar	25

Example 3: Melt multiple id_vars

```
pd.melt(df, id_vars=['Region', 'Name'])
```

	Region	Name	variable	value
0	South	A	Jan	10
1	North	B	Jan	5
2	South	A	Feb	20
3	North	B	Feb	15
4	South	A	Mar	30
5	North	B	Mar	25

Example 4: Melt without id_vars (rare but useful)

	variable	value
0	Region	South
1	Region	North
2	Name	A
3	Name	B
4	Jan	10
5	Jan	5
6	Feb	20
7	Feb	15
8	Mar	30
9	Mar	25

Example 5: Melt with multiple value columns (Advanced beginner level)

```
   Name  Year  Sales_Q1  Sales_Q2
0     A  2023       120       150
1     B  2023       100       130
2     C  2024       140       160
```

```
pd.melt(df, id_vars=['Name','Year'], value_vars=['Sales_Q1','Sales_Q2'])
```

	Name	Year	variable	value
0	A	2023	Sales_Q1	120
1	B	2023	Sales_Q1	100
2	C	2024	Sales_Q1	140
3	A	2023	Sales_Q2	150
4	B	2023	Sales_Q2	130
5	C	2024	Sales_Q2	160

MELT = convert columns into rows

- id_vars → keep these columns
- value_vars → columns to unpivot
- var_name → new column containing old column names
- value_name → new column containing the values

Apply () Function in Pandas

- Apply () is used to apply your own custom function to either:
- A Series → element-wise operations
- A Data Frame → operate on rows or columns
- It is mainly used when built-in Pandas functions are not enough.

Apply Function on Series:

Syntax:

```
df['col'].apply(function)
```

Table:

```
   first_name last_name  age  salary department
0       Rahul     Sharma   25   45000      Sales
1       Anita        Rao   32   55000         HR
2       Kiran      Patil   19   30000         IT
3       Sneha       Nair   28   60000    Finance
```

```
df["Double_Salary"] = df['salary'].apply(lambda x: x*2)
df
```

	first_name	last_name	age	salary	department	Double_Salary
0	Rahul	Sharma	25	45000	Sales	90000
1	Anita	Rao	32	55000	HR	110000
2	Kiran	Patil	19	30000	IT	60000
3	Sneha	Nair	28	60000	Finance	120000

Apply Function on Data Frame:

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

Method 1:

```
df['full_name'] = df.apply(lambda row: row['first_name'] + " " + row['last_name'], axis=1)
df
```

	first_name	last_name	age	salary	department	Double_Salary	full_name
0	Rahul	Sharma	25	45000	Sales	90000	Rahul Sharma
1	Anita	Rao	32	55000	HR	110000	Anita Rao
2	Kiran	Patil	19	30000	IT	60000	Kiran Patil
3	Sneha	Nair	28	60000	Finance	120000	Sneha Nair

Method 2:

```
def make_full_name(row):
    return row['first_name'] + " " + row['last_name']

df['full_name1'] = df.apply(make_full_name, axis=1)
```

```
df
```

	first_name	last_name	age	salary	department	Double_Salary	full_name	full_name1
0	Rahul	Sharma	25	45000	Sales	90000	Rahul Sharma	Rahul Sharma
1	Anita	Rao	32	55000	HR	110000	Anita Rao	Anita Rao
2	Kiran	Patil	19	30000	IT	60000	Kiran Patil	Kiran Patil
3	Sneha	Nair	28	60000	Finance	120000	Sneha Nair	Sneha Nair

When to use?

- Combining multiple columns
- Creating complex derived columns
- Row-level logic (if-else based on multiple columns)

The Most Important Parameter (axis):

axis	meaning	when to use
axis=0 (default)	apply function to **each column**	when summarizing columns
axis=1	apply function to **each row**	when combining multiple columns

---

Common Edge Cases:

Edge Case 1: Using apply for simple math (slow)

```
df['new'] = df['price'].apply(lambda x: x * 2)   # slow
df['new'] = df['price'] * 2                      # fast → preferred
```

Edge Case 2: axis=1 is slow because it loops rows

```
df.apply(lambda row: ..., axis=1)
```

This is slow on large datasets (100k+ rows).

Edge Case 3: Returning lists gives unexpected columns

```
df['col'].apply(lambda x: [x, x*2])
```

You get a column of lists, **not** two separate columns.

To expand:

python

```
df[['a','b']] = df['col'].apply(lambda x: pd.Series([x, x*2]))
```

Edge Case 4: apply () with if-else on multiple columns

Don't do this for simple conditions:

```python
df['flag'] = df.apply(lambda r: 1 if r['age'] > 18 else 0, axis=1)
```

Use vectorized:

```python
df['flag'] = (df['age'] > 18).astype(int)
```

Edge Case 5: apply () with missing values (NaN)

If your function assumes valid data:

```python
df['col'].apply(lambda x: x.lower())   # error if x is NaN
```

Fix:

```python
df['col'].apply(lambda x: x.lower() if pd.notnull(x) else x)
```

Edge Case 6: apply() modifies a copy, not the original

This fails silently:

```python
df['col'].apply(lambda x: x + 1)
```

It doesn't change df unless you assign back:

```python
df['col'] = df['col'].apply(lambda x: x + 1)
```

Practical mini-cheat sheet

✔ Clean text

```python
df['name'] = df['name'].apply(lambda x: x.strip().title())
```

✔ Extract length

```python
df['name_length'] = df['name'].apply(len)
```

Create new column from multiple columns

python

```python
df['full'] = df.apply(lambda r: f"{r['first']} {r['last']}", axis=1)
```

Custom binning

python

```python
df['category'] = df['value'].apply(lambda x: 'high' if x > 100 else 'low')
```

Handle complex logic

python

```python
def tax(row):
    if row['country'] == 'IN':
        return row['price'] * 0.18
    else:
        return row['price'] * 0.10

df['tax'] = df.apply(tax, axis=1)
```

Map () Function in Pandas

- Map () is used **only on Series** (columns), not on Data Frames.

It is mainly used for:

- Replacing values
- Cleaning categorical columns
- Converting text labels to numbers
- Doing simple value-to-value transformation
- Think of map () like a dictionary lookup for a column

The 3 Most Important Ways to Use map ()

(A) 1. Using map () with a dictionary (MOST IMPORTANT USE-CASE)

- You provide a dictionary that tells Pandas how to convert each value.

	Name	Gender	Status	City	Score
0	Raksha	Female	Active	Bengaluru	85
1	Sathya	Male	Inactive	Udupi	72
2	Ananya	Female	Active	Mangaluru	90
3	Rohan	Male	Inactive	Bengaluru	60
4	Megha	Female	Active	Udupi	78

```
df['Gender_code'] = df['Gender'].map({'Male': 1, 'Female': 0})
df
```

	Name	Gender	Status	City	Score	Gender_code
0	Raksha	Female	Active	Bengaluru	85	0
1	Sathya	Male	Inactive	Udupi	72	1
2	Ananya	Female	Active	Mangaluru	90	0
3	Rohan	Male	Inactive	Bengaluru	60	1
4	Megha	Female	Active	Udupi	78	0

This is the most common usage — converting categories to numbers.

Why useful?

- Used in data cleaning
- ML preprocessing
- Dashboard/category mapping
- Standardizing inconsistent values

(B) 2. Using map () with a function

```
df['name_upper'] = df['Name'].map(lambda x: x.upper())
df
```

	Name	Gender	Status	City	Score	Gender_code	name_upper
0	Raksha	Female	Active	Bengaluru	85	0	RAKSHA
1	Sathya	Male	Inactive	Udupi	72	1	SATHYA
2	Ananya	Female	Active	Mangaluru	90	0	ANANYA
3	Rohan	Male	Inactive	Bengaluru	60	1	ROHAN
4	Megha	Female	Active	Udupi	78	0	MEGHA

- Use this when you want a **simple transformation** on each element.

Tables:

df_customers

	customer_id	name	city
0	101	Asha	Bangalore
1	102	Manoj	Mumbai
2	103	Priya	Delhi
3	104	Ravi	Bangalore

df_city_lookup

	city	city_code
0	Bangalore	1011
1	Mumbai	2011
2	Delhi	3011

```
city_lookup_series = df_city_lookup.set_index('city')['city_code']
city_lookup_series
```

```
city
Bangalore    1011
Mumbai       2011
Delhi        3011
Name: city_code, dtype: int64
```

```
df_customers['city_code'] = df_customers['city'].map(city_lookup_series)
```

```
df_customers
```

	customer_id	name	city	city_code
0	101	Asha	Bangalore	1011
1	102	Manoj	Mumbai	2011
2	103	Priya	Delhi	3011
3	104	Ravi	Bangalore	1011

Map v/s Replace

Feature	map ()	replace ()
Works only on Series	Yes	Yes
Unmapped values become NaN	Yes	No (remain unchanged)
Supports functions	Yes	No
For simple replacements	Not ideal	Best

Example1:

```
import pandas as pd

data = {
    'EmployeeID': [101, 102, 103, 104, 105],
    'Name': ['Rohan', 'Priya', 'Amit', 'Sneha', 'Vikas'],
    'Department': ['HR', 'IT', 'Finance', 'IT', 'HR'],
    'Gender': ['M', 'F', 'M', 'F', 'M'],
    'Salary': [50000, 65000, 70000, 62000, 55000],
    'PerformanceScore': [3, 4, 5, 3, 4]
}

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

```
   EmployeeID   Name Department Gender  Salary  PerformanceScore
0         101  Rohan         HR      M   50000                 3
1         102  Priya         IT      F   65000                 4
2         103   Amit    Finance      M   70000                 5
3         104  Sneha         IT      F   62000                 3
4         105  Vikas         HR      M   55000                 4
```

```
dictionary = {"M":"Male","F":"Female"}
df["Gender"] = df["Gender"].map(dictionary)
```

```
df
```

	EmployeeID	Name	Department	Gender	Salary	PerformanceScore
0	101	Rohan	HR	Male	50000	3
1	102	Priya	IT	Female	65000	4
2	103	Amit	Finance	Male	70000	5
3	104	Sneha	IT	Female	62000	3
4	105	Vikas	HR	Male	55000	4

Example 2: Using Lambda Function

```
df['PerformanceStatus'] = df['PerformanceScore'].map(
    lambda x: 'Pass' if x >= 4 else 'Fail'
)
```

```
df
```

	EmployeeID	Name	Department	Gender	Salary	PerformanceScore	PerformanceStatus
0	101	Rohan	HR	Male	50000	3	Fail
1	102	Priya	IT	Female	65000	4	Pass
2	103	Amit	Finance	Male	70000	5	Pass
3	104	Sneha	IT	Female	62000	3	Fail
4	105	Vikas	HR	Male	55000	4	Pass

Example 3:

```
df['Dept_Upper'] = df['Department'].map(lambda x: x.upper())
```

```
df
```

	EmployeeID	Name	Department	Gender	Salary	PerformanceScore	PerformanceStatus	Dept_Upper
0	101	Rohan	HR	Male	50000	3	Fail	HR
1	102	Priya	IT	Female	65000	4	Pass	IT
2	103	Amit	Finance	Male	70000	5	Pass	FINANCE
3	104	Sneha	IT	Female	62000	3	Fail	IT
4	105	Vikas	HR	Male	55000	4	Pass	HR

Example 4: Using Custom Function (Not Lambda)

```
def grade(score):
    if score == 5:
        return 'Excellent'
    elif score == 4:
        return 'Good'
    else:
        return 'Average'

df['Grade'] = df['PerformanceScore'].map(grade)
```

```
df
```

	EmployeeID	Name	Department	Gender	Salary	PerformanceScore	PerformanceStatus	Dept_Upper	Grade
0	101	Rohan	HR	Male	50000	3	Fail	HR	Average
1	102	Priya	IT	Female	65000	4	Pass	IT	Good
2	103	Amit	Finance	Male	70000	5	Pass	FINANCE	Excellent
3	104	Sneha	IT	Female	62000	3	Fail	IT	Average
4	105	Vikas	HR	Male	55000	4	Pass	HR	Good

Example difference:

```python
df['gender'].map({'Male': 1})
```

All values **not** in dictionary become **NaN**.

But:

```python
df['gender'].replace({'Male': 1})
```

Other values remain the same.

Different Scenarios:

	Name	Gender	Status	City	Score
0	Raksha	Female	Active	Bengaluru	85
1	Sathya	Male	Inactive	Udupi	72
2	Ananya	Female	Active	Mangaluru	90
3	Rohan	Male	Inactive	Bengaluru	60
4	Megha	Female	Active	Udupi	78

Scenario 1: Convert Categories into Numbers

```
df['Gender_code'] = df['Gender'].map({'Female': 1, 'Male': 0})
```

```
df
```

	Name	Gender	Status	City	Score	Gender_code
0	Raksha	Female	Active	Bengaluru	85	1
1	Sathya	Male	Inactive	Udupi	72	0
2	Ananya	Female	Active	Mangaluru	90	1
3	Rohan	Male	Inactive	Bengaluru	60	0
4	Megha	Female	Active	Udupi	78	1

Scenario 2: Replace Inconsistent Labels

	customer_id	name	response	age
0	101	Asha	Yes	23
1	102	Manoj	No	30
2	103	Priya	Y	27
3	104	Ravi	N	25
4	105	Kiran	Maybe	29
5	106	Neha	Yes	24

```
clean_map = {
    'Yes': 1, 'Y': 1, 'No': 0, 'N': 0
}
df['response_clean'] = df['response'].map(clean_map)
df
```

	customer_id	name	response	age	name_length	response_clean
0	101	Asha	Yes	23	4	1.0
1	102	Manoj	No	30	5	0.0
2	103	Priya	Y	27	5	1.0
3	104	Ravi	N	25	4	0.0
4	105	Kiran	Maybe	29	5	NaN
5	106	Neha	Yes	24	4	1.0

Scenario 3: Create a New Feature

```
df['name_length'] = df['name'].map(lambda x: len(x))
df
```

	customer_id	name	response	age	name_length	response_clean
0	101	Asha	Yes	23	4	1.0
1	102	Manoj	No	30	5	0.0
2	103	Priya	Y	27	5	1.0
3	104	Ravi	N	25	4	0.0
4	105	Kiran	Maybe	29	5	NaN
5	106	Neha	Yes	24	4	1.0

applymap () Function in Pandas

- applymap() is used **ONLY on DataFrames**, not on Series.
- 👉 It applies a **function to every single cell** (element) of the DataFrame.
- If you want to transform **every value** in a DataFrame, applymap() is the correct choice.
- applymap() is a **DataFrame-level function** used to apply a **custom function to each and every cell** in the DataFrame.

Syntax:

```
df.applymap(function)
```

Situation	Example
You want to change every value in the DataFrame	Convert all values to string
You want element-wise formatting	Add "₹" to all amounts
You want to apply simple operations to all cells	Find length of each cell
Clean messy data in all columns	Trim whitespaces everywhere

1. applymap () applies a function to *each cell* of the entire Data Frame.

Example 1:

```
Original DataFrame:
   column1  column2  column3 Column4
0        1        4        7       a
1        2        5        8       B
2        3        6        9       D
```

```
# applymap operation
df_multiplied = df.applymap(lambda x: x * 10)

print("\nAfter applymap(lambda x: x * 10):")
print(df_multiplied)
```

```
After applymap(lambda x: x * 10):
   column1  column2  column3     Column4
0       10       40       70  aaaaaaaaaa
1       20       50       80  BBBBBBBBBB
2       30       60       90  DDDDDDDDDD
```

- This multiplies every value inside the DataFrame by 10.
- Even if the DataFrame has 5 columns and 100 rows → it touches all 500 cells.

Example 2:

	power_level	uniform color	species
Goku	12000	orange	saiyan
Vegeta	16000	blue	saiyan
Nappa	4000	black	saiyan
Gohan	1500	orange	half saiyan
Piccolo	3000	purple	namak
Tien	2000	green	human
Yamcha	1600	orange	human
Krillin	2000	orange	human

```
# Apply a function to ALL elements in a Data Frame

def str_len(x):
    return(len(str(x)))

d2 = data.applymap(str_len)
d2
```

```
# If all data is numeric, math operations are broadcast elementwise by default!

d2 ** 2
```

	power_level	uniform color	species
Goku	25	36	36
Vegeta	25	16	36
Nappa	16	25	36
Gohan	16	36	121
Piccolo	16	36	25
Tien	16	25	25
Yamcha	16	36	25
Krillin	16	36	25

---

applymap () is mainly used for formatting or cleaning values:

Typical use cases:

✔ **Formatting numbers**

python

```
df.applymap(lambda x: f"{x:.2f}")
```

✔ **Removing whitespace**

python

```
df.applymap(lambda x: x.strip() if isinstance(x, str) else x)
```

Converting type

```python
df.applymap(lambda x: int(x))
```

Checking conditions

```python
df.applymap(lambda x: "High" if x > 50 else "Low")
```

- applymap () is SLOW for very large Data Frames.
- Because it processes each cell one-by-one, it is slower than vectorized operations.
- On big data, avoid applymap unless necessary.

---

4. When to Use What? (Simple Rule)

- `map()` → **Series (one column)** → for element-wise operations.
- `apply()` → **Both Series & DataFrame** → for row-wise/column-wise or custom function logic.
- `applymap()` → **DataFrame only** → for element-wise operation on all cells.

5. Quick Comparison Table

Function	Works On	Use Case Example
`map()`	Series	Replace gender codes, transform a single column
`apply()`	Series/DF	Row-wise profit calculation, column sums
`applymap()`	DataFrame	Apply formatting (e.g., uppercase) to all cells

---

.dt Accessor

- In Pandas, the .dt accessor is used to work with date and time values that are stored inside a datetime column or Series.
- 👉 Think of .dt as a "date-time toolbox" that helps you extract parts like year, month, day, weekday, etc., from your datetime data.

Example:

Let's take an example:

```python
import pandas as pd

# Creating a sample date column
data = {'Date': ['2024-01-05', '2024-02-15', '2024-03-25']}
df = pd.DataFrame(data)

# Convert 'Date' column to datetime type
df['Date'] = pd.to_datetime(df['Date'])

print(df)
```

Output:

```yaml
        Date
0 2024-01-05
1 2024-02-15
2 2024-03-25
```

🔹 Common `.dt` Attributes

Attribute	Meaning	Example Output
`.dt.year`	Extracts the **year**	`[2024, 2024, 2024]`
`.dt.month`	Extracts the **month number (1–12)**	`[1, 2, 3]`
`.dt.day`	Extracts the **day of the month**	`[5, 15, 25]`
`.dt.weekday`	Returns **day of the week (0=Monday, 6=Sunday)**	`[4, 3, 0]`
`.dt.day_name()`	Returns **day name**	`['Friday', 'Thursday', 'Monday']`
`.dt.month_name()`	Returns **month name**	`['January', 'February', 'March']`
`.dt.quarter`	Returns **quarter of the year (1–4)**	`[1, 1, 1]`
`.dt.is_month_start`	Checks if date is **first day of month**	`[False, False, False]`
`.dt.is_month_end`	Checks if date is **last day of month**	`[False, False, False]`

🔹 Example in Use

```python
df['Year'] = df['Date'].dt.year
df['Month'] = df['Date'].dt.month
df['Day'] = df['Date'].dt.day
df['Day_Name'] = df['Date'].dt.day_name()
df['Quarter'] = df['Date'].dt.quarter
print(df)
```

Output:

```yaml
        Date  Year  Month  Day  Day_Name  Quarter
0 2024-01-05  2024      1    5    Friday        1
1 2024-02-15  2024      2   15  Thursday        1
2 2024-03-25  2024      3   25    Monday        1
```

🔹 `.dt` also works with time data

If you have time values:

```python
df['Time'] = pd.to_datetime(['2024-01-05 14:35:20', '2024-02-15 09:10:05', '2024-03-25 23:45:55'])
df['Hour'] = df['Time'].dt.hour
df['Minute'] = df['Time'].dt.minute
df['Second'] = df['Time'].dt.second
print(df[['Time', 'Hour', 'Minute', 'Second']])
```

Output:

```sql
                 Time  Hour  Minute  Second
0 2024-01-05 14:35:20    14      35      20
1 2024-02-15 09:10:05     9      10       5
2 2024-03-25 23:45:55    23      45      55
```