Pandas

1. What is Data Manipulation and Analysis?

Data Manipulation: Changing, organizing, or cleaning raw data to make it usable.
Data Analysis: Studying the data to find patterns, trends, or insights.

---

2. Who Created Pandas and Why?

- **Created by**: **Wes McKinney** in **2008**.
- **Why**: He needed a powerful and flexible tool for analyzing financial data in Python. Excel and existing tools weren't enough.

---

3. What is Pandas?

- **Pandas** is a **Python library** used for data manipulation and analysis.
- It provides **data structures** like **Series** (1D) and **DataFrame** (2D table) to work easily with data.

---

4. What Makes Pandas Unique?

- Easy to use and powerful data structures (like Excel in Python).
- Handles large datasets efficiently.
- Supports data cleaning, filtering, grouping, merging, etc.
- Works well with other libraries like NumPy and Matplotlib.

1. Series

- A **Series** is like a **single column** of data.

- It's **1-dimensional** (1D).
- Think of it like a **list with labels (index)**.

🔹 **Example:**

```
import pandas as pd

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

Output:

0   10
1   20
2   30
dtype: int64

👉 Index (0,1,2) on the left and values (10,20,30) on the right.

---

2. DataFrame

- A **DataFrame** is like a **table or Excel sheet**.
- It's **2-dimensional** (rows and columns).
- It can have **multiple columns**, each like a Series.

🔹 **Example:**

```
data = {'Name': ['John', 'Alice'], 'Age': [25, 30]}
df = pd.DataFrame(data) # index=False
print(df)
```

Output:

	Name	**Age**
0	**John**	**25**
1	**Alice**	**30**

👉 Rows + Columns = DataFrame

Feature	Series	DataFrame
Dimension	1D	2D
Structure	Like a column	Like a table
Usage	Simple lists	Complex datasets

✅ 1. Read Data in Pandas

📄 Read CSV file:

```
import pandas as pd
# Read a CSV file
df = pd.read_csv('filename.csv')
```

📊 Read Excel file:

```
df = pd.read_excel('filename.xlsx')
```

📝 Read from a dictionary:

```
data = {'Name': ['John', 'Alice'], 'Age': [25, 30]}
df = pd.DataFrame(data)
```

💾 2. Save Data in Pandas

📄 Save to CSV:

```
df.to_csv('output.csv', index=False)
```

📊 Save to Excel:

```
df.to_excel('output.xlsx', index=False)
```

💡 Notes:

- index=False means: Don't save the row numbers.
- Make sure the file path is correct if you're working with folders.

✅ 1. head()

- Shows the **first 5 rows** by default.
- You can also specify how many rows to show.

🔹 **Example:**

```
df.head()       # Shows first 5 rows
df.head(3)      # Shows first 3 rows
```

✅ 2. tail()

- Shows the **last 5 rows** by default.
- You can also choose how many rows to view.

🔹 **Example:**

```
df.tail()       # Shows last 5 rows
df.tail(2)      # Shows last 2 rows
```

💡 Use:

Helpful to **quickly check your data**—either the top (start) or bottom (end) of the DataFrame.

✅ df.info()

This function gives you a **summary** of your DataFrame.

🔹 What does it show?

- Total number of rows and columns
- Column names
- Non-null (non-empty) values
- Data types (int, float, object, etc.)
- Memory usage

🔍 Example:

```
import pandas as pd

data = {'Name': ['Alice', 'Bob', 'Charlie'],
        'Age': [25, 30, None]}

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

Output:

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 3 entries, 0 to 2
Data columns (total 2 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   Name    3 non-null      object
 1   Age     2 non-null      float64
dtypes: float64(1), object(1)
memory usage: 176.0 bytes
```

💡 Why use .info()?

- To **check for missing data**

- To **understand data types**
- To know the **structure** of the dataset

✅ df.describe()

It gives **statistical summary** of your **numeric columns**.

🔹 What does it show?

- **Count** – Total non-null (non-empty) entries
- **Mean** – Average
- **Std** – Standard deviation (spread of data)
- **Min** – Minimum value
- **25%** – 1st quartile (25% of data is below this)
- **50%** – Median (middle value)
- **75%** – 3rd quartile (75% of data is below this)
- **Max** – Maximum value

🔍 Example:

```
import pandas as pd

data = {'Age': [25, 30, 35, 40, 45]}

df = pd.DataFrame(data)

print(df.describe())
```

Output:

```
        Age
count   5.000000
mean   35.000000
std     7.905694
min    25.000000
25%    30.000000
50%    35.000000
75%    40.000000
max    45.000000
```

💡 Use:

- To get a **quick understanding** of your data's distribution.
- Very useful for **data analysis** and **outlier detection**.

✅ 1. df.shape

- Shows the **number of rows and columns** in the DataFrame.
- Returns a **tuple**: (rows, columns)

🔹 **Example:**

```
df.shape
```

Output:

(5, 3) # Means 5 rows and 3 columns

✅ 2. df.columns

- Shows the **names of all columns** in the DataFrame.
- Returns an **Index object** (like a list of column names)

🔹 **Example:**

```
df.columns
```

Output:

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

💡 Use:

- .shape → To know the **size of your data**
- .columns → To know what **columns are available** for analysis

🔹 Selecting Columns

✅ 1. Select a Single Columns by Name

```
price=df["Price"]
print(price)
```

✅ 2. Select Multiple Columns by Name

```
subset = df[["Name","Salary"]]
print("\n subset with Name and Salary")
print(subset)
```

🔹 Selecting with Conditions

Assume this example:

```
import pandas as pd
```

```
data = {'Name': ['Alice', 'Bob', 'Charlie', 'David'],

        'Age': [25, 30, 35, 40],

        'City': ['Delhi', 'Mumbai', 'Delhi', 'Chennai']}

df = pd.DataFrame(data)
```

✅ 3. Single Condition

```
df[df['Age'] > 30]    # Rows where Age is greater than 30
```

✅ 4. Multiple Conditions (AND)

```
df[(df['Age'] > 30) & (df['City'] == 'Delhi')]
```

👉 Use & for **AND**, and wrap each condition in **( )**

✅ 5. Multiple Conditions (OR)

```
df[(df['Age'] < 30) | (df['City'] == 'Chennai')]
```

👉 Use | for **OR**

✅ 6. Selecting Specific Columns After Filtering

```
df[df['Age'] > 30][['Name', 'City']]
```

✅ 1. Add New Column Normally

```
df['New_Column'] = [value1, value2, value3, ...]
```

Example:

```
df['Country'] = ['India', 'USA', 'China', 'Japan']
```

✅ 2. Add New Column at Specific Position using insert()

```
df.insert(loc=1, column='Gender', value=['F', 'M', 'M', 'M'])
```

- loc=1 → position (starts from 0)
- column='Gender' → new column name
- value=[...] → values for the column

✅ Example Code:

```
import pandas as pd

data = {'Name': ['Alice', 'Bob', 'Charlie'],

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

df = pd.DataFrame(data)

df.insert(loc=1, column='Gender', value=['F', 'M', 'M'])

print(df)
```

Output:

```
      Name Gender  Age
0    Alice      F   25
1      Bob      M   30
2  Charlie      M   35
```

✅ 1. Update a Specific Cell

Use df.at[] or df.loc[]:

🔹 **Using .at[] (fast for single cell):**

```
df.at[2, 'Age'] = 40    # Update value in row index 2, column 'Age'
```

🔹 **Using .loc[]:**

```
df.loc[2, 'Name'] = 'Charlie Updated'
```

✅ 2. Update Multiple Cells in a Row

```
df.loc[1] = ['Robert', 32, 'M']   # Updates entire row at index 1
```

Be sure your values match the column order.

✅ Example:

```
import pandas as pd

data = {'Name': ['Alice', 'Bob', 'Charlie'], 'Age': [25, 30, 35], 'Gender': ['F', 'M', 'M']}
df = pd.DataFrame(data)

df.at[1, 'Age'] = 31            # Update Bob's age
df.loc[2, 'Name'] = 'Charles'     # Update Charlie's name

print(df)
```

Output:

```
      Name  Age Gender
0    Alice   25      F
1      Bob   31      M
2  Charles   35      M
```

✅ 1. Replace Entire Column with New Values

```
df['Age'] = [26, 31, 36]   # Replace all values in 'Age' column
```

The number of values must match the number of rows.

✅ 2. Update Column Using a Formula

```
df['Age'] = df['Age'] + 5   # Increase each age by 5
```

✅ 3. Update Using Conditions

```
df['Status'] = 'Active'                    # New column for all rows
df.loc[df['Age'] > 30, 'Status'] = 'Senior'  # Update where Age > 30
```

✅ Example:

```python
import pandas as pd

data = {'Name': ['Alice', 'Bob', 'Charlie'], 'Age': [25, 30, 35]}
df = pd.DataFrame(data)

df['Age'] = df['Age'] + 2   # Add 2 years to everyone's age

print(df)
```

Output:

```
      Name  Age
0    Alice   27
1      Bob   32
2  Charlie   37
```

.drop() method in **Pandas** is used to **remove rows or columns** from a DataFrame.

✅ Syntax:

```python
df.drop(labels, axis, inplace=False)
```

✅ 1. Drop Column

```python
df.drop('Age', axis=1)   # axis=1 means column
```

To apply it permanently:

```python
df.drop('Age', axis=1, inplace=True)
```

✅ 2. Drop Multiple Columns

```python
df.drop(['Age', 'Gender'], axis=1, inplace=True)
```

✅ 3. Drop Row by Index

```python
df.drop(1, axis=0)   # axis=0 means row
```

Drop multiple rows:

```
df.drop([0, 2], axis=0, inplace=True)
```

✅ Example:

```
import pandas as pd

data = {'Name': ['Alice', 'Bob', 'Charlie'], 'Age': [25, 30, 35], 'Gender': ['F', 'M', 'M']}
df = pd.DataFrame(data)

df.drop('Gender', axis=1, inplace=True)  # Drop the Gender column
print(df)
```

Output:

```
      Name  Age
0    Alice   25
1      Bob   30
2  Charlie   35
```

✅ 1. Check for Missing Values

```
df.isnull()        # Shows True where values are missing
df.isnull().sum()  # Total missing values in each column
```

✅ 2. Drop Missing Values

🔹 Drop rows with missing values:

```
df.dropna(inplace=True)
```

🔹 Drop columns with missing values:

```
df.dropna(axis=1, inplace=True)
```

✅ 3. Fill Missing Values

🔹 **Fill with a fixed value:**

```
df.fillna(0, inplace=True)   # Replace NaN with 0
```

🔹 **Fill with column mean/median/mode:**

```
df['Age'].fillna(df['Age'].mean(), inplace=True)
```

✅ 4. Example:

```
import pandas as pd
import numpy as np

data = {'Name': ['Alice', 'Bob', 'Charlie'],
        'Age': [25, np.nan, 35],
        'Gender': ['F', 'M', np.nan]}

df = pd.DataFrame(data)

df['Age'].fillna(df['Age'].mean(), inplace=True)
df['Gender'].fillna('Unknown', inplace=True)

print(df)
```

Output:

```
      Name   Age   Gender
0    Alice  25.0        F
1      Bob  30.0        M
2  Charlie  35.0  Unknown
```

🔹 **What is interpolate()?**

interpolate() is used to **fill missing (NaN) values** in a DataFrame or Series by estimating them using other values.

✅ Linear Interpolation (Default Method)

It fills missing values by **assuming a straight line** between the two known points.

🔹 **Example:**

```
import pandas as pd
import numpy as np

data = {'Score': [10, np.nan, np.nan, 40]}
df = pd.DataFrame(data)

df['Score'] = df['Score'].interpolate(method='linear')

print(df)
```

🔸 **Output:**

```
   Score
0   10.0
1   20.0
2   30.0
3   40.0
```

🔍 It filled 20 and 30 by calculating evenly between 10 and 40.

✅ You can also interpolate along columns:

```
df.interpolate(method='linear', axis=1)
```

✅ 1. Sort by One Column

Use sort_values():

```
df.sort_values('Age')          # Ascending by default
df.sort_values('Age', ascending=False)   # Descending
```

✅ 2. Sort by Multiple Columns

```
df.sort_values(['Gender', 'Age'])   # Sort by Gender, then Age
```

🔹 Descending and ascending together:

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

✅ Example:

```
import pandas as pd

data = {
    'Name': ['Alice', 'Bob', 'Charlie', 'David'],
    'Age': [25, 30, 25, 35],
    'Gender': ['F', 'M', 'M', 'M']
}

df = pd.DataFrame(data)

# Sort by Age (ascending)
df1 = df.sort_values('Age')

# Sort by Gender (asc) and Age (desc)
df2 = df.sort_values(['Gender', 'Age'], ascending=[True, False])

print("Sorted by Age:\n", df1)
print("\nSorted by Gender & Age:\n", df2)
```

✅ Output:

```
Sorted by Age:
     Name  Age Gender
0   Alice   25      F
2 Charlie   25      M
1     Bob   30      M
3   David   35      M
```

```
Sorted by Gender & Age:
      Name  Age Gender
0    Alice   25      F
3    David   35      M
1      Bob   30      M
2  Charlie   25      M
```

✅ Summary Function :

```python
import pandas as pd

data = {'Name': ['Alice', 'Bob', 'Charlie'], 'Age': [25, 30, 35]}
df = pd.DataFrame(data)

print(df['Age'].mean())  # Output: 30.0
print(df['Age'].max())   # Output: 35
print(df['Age'].min())   # Output: 25
print(df['Age'].sum())   # Output: 90
print(df['Age'].count()) # Output: 3
print(df.describe())     # Summary stats
```

✅ 1. Single Column Grouping

Group by one column and apply a function:

```python
df.groupby('Department')['Salary'].mean()
```

Groups rows by Department and returns average Salary.

✅ 2. Multiple Column Grouping

Group by more than one column:

```
df.groupby(['Department', 'Gender'])['Salary'].sum()
```

Groups by both Department and Gender, then totals Salary.

✅ 3. Use Multiple Functions

```
df.groupby('Department')['Salary'].agg(['mean', 'sum', 'max'])
```

✅ Example:

```
import pandas as pd

data = {
    'Name': ['A', 'B', 'C', 'D', 'E'],
    'Department': ['HR', 'IT', 'HR', 'IT', 'HR'],
    'Gender': ['F', 'M', 'M', 'F', 'F'],
    'Salary': [30000, 40000, 32000, 45000, 31000]
}

df = pd.DataFrame(data)

# Group by Department
print(df.groupby('Department')['Salary'].mean())

# Group by Department and Gender
print(df.groupby(['Department', 'Gender'])['Salary'].sum())

# Group with multiple functions
print(df.groupby('Department')['Salary'].agg(['mean', 'max', 'min']))
```

✅ Output:

Department

Vikas

```
HR    31000.0
IT    42500.0
Name: Salary, dtype: float64

Department  Gender
HR          F         61000
            M         32000
IT          F         45000
            M         40000
Name: Salary, dtype: int64

           mean   max   min
Department
HR        31000  32000  30000
IT        42500  45000  40000
```

✅ 1. Basic Merge

```
pd.merge(df1, df2, on='ID')
```

Merges both DataFrames using a **common column** (ID here).

✅ 2. Types of Merge (JOINs)

```
pd.merge(df1, df2, on='ID', how='inner')   # Only matching rows
pd.merge(df1, df2, on='ID', how='left')    # All from df1
pd.merge(df1, df2, on='ID', how='right')   # All from df2
pd.merge(df1, df2, on='ID', how='outer')   # All from both
```

✅ Example:

```
import pandas as pd

df1 = pd.DataFrame({
    'ID': [1, 2, 3],
```

```
    'Name': ['Alice', 'Bob', 'Charlie']
})

df2 = pd.DataFrame({
    'ID': [2, 3, 4],
    'Score': [90, 85, 88]
})

# Inner Merge (only common IDs)
merged = pd.merge(df1, df2, on='ID', how='inner')
print(merged)
```

Output:

```
  ID   Name  Score
0  2    Bob     90
1  3  Charlie   85
```

✅ Merge on different column names:

```
pd.merge(df1, df2, left_on='ID1', right_on='ID2')
```

✅ 1. Concatenate Vertically (Row-wise)

```
pd.concat([df1, df2])
```

- Stacks **df2 below df1** (like UNION in SQL)
- Indexes are kept unless ignore_index=True

Example:

```
df1 = pd.DataFrame({'ID': [1, 2], 'Name': ['Alice', 'Bob']})
df2 = pd.DataFrame({'ID': [3, 4], 'Name': ['Charlie', 'David']})
```

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

✅ 2. Concatenate Horizontally (Column-wise)

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

- Joins DataFrames **side by side** (like adding new columns)

✅ 3. Add Keys (MultiIndex)

```
pd.concat([df1, df2], keys=['Batch1', 'Batch2'])
```

Useful for keeping track of which part came from which DataFrame.

🔸 **Notes:**

- Use axis=0 for rows (default)
- Use axis=1 for columns
- Make sure columns match when stacking row-wise