Pandas

What is Pandas?

Pandas is one of the most widely used Python libraries for **data analysis and manipulation**. It is open-source and built on the top of **NumPy**. It adds high-level data structures and tools that make it easier to work with **tabular**, **labeled**, or **heterogeneous** datasets.

Data Structures

Pandas introduces two important data structure: Series & DataFrame.

Usage

```
import pandas as pd

df = pd.DataFrame({
    "Name": ["Alice", "Bob"],
    "Cgpa": [9.5, 8.7]
})
```

Core Data Structures in Pandas

Series

A Series is a **one-dimensional labeled array** (like a column in a spreadsheet). It can hold data of any type: integers, floats, strings, Python objects.

It has two main components:

1. **Values** - the actual data
2. **Index** - labels for each value

Usage

```python
s = pd.Series([23, 24, 25, 26])
print(s)
print(type(s))

# Indexing
print(s[0])      # 23
print(s[2])      # 25

print(s.index)       # all labels
```

Characteristics of a Series

1. They are Homogeneous - store one type of data.
2. They support Vectorized operations.
3. They can handle missing values with NaN.
4. They have mutable values but immutable size.

```python
# Custom Indexing
s2 = pd.Series([23, 24, 25, 26], index = ["Adam", "Eve", "Charlie", "Bob"])
print(s2["Eve"])    # 24
print(s2["Bob"])    # 26

# Vectorized Operations
s1 = pd.Series([1, 2, 3])
s2 = pd.Series([4, 5, 6])

print(s1 + s2)

# Mutable Values but immutable size
s = pd.Series([1, 2, 3, 4, 5])
s[0] = 100

print(s)

changed_s = s.drop(1)
print(changed_s)
print(s)
```


DataFrame

A DataFrame is a **two-dimensional, tabular data structure** (like a spreadsheet or SQL table).

It consists of:

- Rows
- Columns
- Index (row labels)
- Column labels

Each column in a DataFrame is a Series.

Usage

```
# Creating DataFrame in pandas - using dictionary
info = {
    "Name" : ["Adam", "Eve", "Bob"],
    "Marks" : [78, 99, 85],
    "Grade" : ['B', 'O', 'A']
}

df = pd.DataFrame(info)

print(df)
print(type(df))

print(df.index)         # row labels
print(df.columns)       # column labels

# Creating DataFrom using Numpy array
np_arr = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
df = pd.DataFrame(np_arr, columns=["Col1", "Col2", "Col3"])
print(df)

# Creating DataFrom using Lists
l = [["Adam", 96], ["Eve", 75], ["Bob", 82], ["Charlie", 92]]
df = pd.DataFrame(l, columns=["Name", "Marks"])
print(df)
```


Working with Data Files

Most of the time when we deal with data we'll get it in a format like **CSV** (Comma-Separated Values) or **JSON** (JavaScript Object Notation) and Pandas makes it extremely easy to read data from various file formats.

Importing Data

```
df_csv = pd.read_csv("data.csv")     # importing data from csv file

df_json = pd.read_json("data.json")  # importing data from json file
```

Exporting Data

```
df.to_csv("output.csv")   # exporting to csv

df.to_csv("output.csv", index=False)  # exporting without index

df.to_json("output.json")   # exporting to json
```


DataFrame Methods

We have a lot of useful methods with DataFrame.

Data Viewing & Inspection

These help us take a quick look at our dataset.

1. `df.head(n)` - Shows the first n rows (default = 5)
2. `df.tail(n)` - Shows the last n rows (default = 5)
3. `df.sample(n)` - Shows random n rows (default = 1)
4. `df.info()` - Displays column names, data types, memory usage
5. `df.describe(n)` - Shows descriptive statistics for numeric columns.
6. `df.nunique(n)` - Shows count of distinct values exist in each column.

We also have attributes like:

1. `df.shape` - Returns (rows, columns).
2. `df.columns` - List of column names.
3. `df.dtypes` - Data types of each column.

```
data = {
    'Name': ['Aarav', 'Isha', 'Rohan', 'Sneha', 'Vikram'],
    'Age': [25, 30, 35, 40, 45],
    'City': ['Delhi', 'Mumbai', 'Bangalore', 'Kolkata', 'Chennai']
}

df = pd.DataFrame(data)

print(df.head())
print(df.tail())
print(df.sample())
print(df.info())
print(df.describe())
print(df.nunique())

print(df.dtypes)
print(df.shape)
print(df.columns)
```

Indexing & Data Selection

- **Selecting Columns**

  We can select by column name (as a Series) or multiple columns at once.

```
df["country"]        # returns a Series
df[["city", "aqi"]]  # returns a DataFrame
```

- **Selecting Rows**

  By index label (`loc`) - Use row labels (if you have custom index) or column names.

```
df.loc[0]             # 1st row (by label)
df.loc[0:3]           # row0 to row3 - both inclusive in loc

df.loc[0, "country"]   # 0th row & specific column
```

  Note : In `loc`, when we slice, the ending index is inclusive.

By index position (`iloc`) - Use integer positions.

```
df.iloc[0]             # 1st row
df.iloc[4:7]           # 1st row

df.iloc[0, 2]          # 1st row & 3rd column (by position)
```

Note : In `iloc`, when we slice, the ending index is NOT inclusive.

- **Selecting single Cell**

  We have `at` & `iat` for fast access to single cells.

```
# Select single scalar value
df.at[0, "city"]
df.iat[0, 2]
```

Filtering & Querying Data

- **Boolean Filtering**

  We can filter a DataFrame using a condition that returns a boolean Series.

```
df[df['Age'] > 30]
df[(df['Age'] > 30) & (df['Salary'] > 50000)]
```

  We can use `&` for AND, `|` for OR & wrap each condition in parentheses `()`.

- `df.query()` **Method**

  Query method provides us with SQL-like filtering. We can pass our query in a string.

```
# Example 1
df.query("Age > 30 and Salary < 70000")

# Example 2
my_country = "IN"
df.query("country == @my_country")
```

  Query returns a COPY, not a VIEW.

Query String Rules:

1. We write the condition in a string.
2. We can use operators like `and` , `or` , `not` , `==` , `!=` , `>` , `<` , `>=` , `<=` etc.
3. Use backticks for column names with spaces or symbols.
4. Use @ for Python variables.

Important - Rule of thumb:

- Direct filtering using df[...] - prefer using `&` / `|` for AND & OR.
- Inside `query()` strings - prefer using `and` / `or` for AND & OR.

Data Cleaning

- **Handle Missing Values (`NaN`)**

1. `df.isnull()` - Shows boolean DataFrame, True where NaN
2. `df.isnull().sum()` - Count of NaNs per column
3. `df.dropna()` - Drop rows with any NaN
4. `df.fillna(val)` - Fills NaN with a value
5. `df.ffill()` - Forward Fill (carry previous value)
6. `df.bfill()` - Backward Fill (carry next value)

```
# Handle Missing values
df.isnull()             # True for null values (NaN)
df.isnull().sum()       # counts missing vals per column

df.dropna()             # drops rows with missing values
df.dropna(axis = 1)     # drops cols with missing values

df.fillna(0)            # fills NaN with 0
df["age"] = df["age"].fillna(df["age"].mean())    # fills column with mean

df.ffill()              # forward fill
df.bfill()              # backward fill
```

- **Handle Duplicate Values (`NaN`)**
  1. `df.duplicated()` - Find duplicates
  2. `df.drop_duplicates()` - Remove duplicate rows

```
df.duplicated()               # True for duplicate rows
df.duplicated("country")      # True for duplicate rows in particular col
df.duplicated(["country", "gender"])

df.drop_duplicates()                # drops duplicate rows
```

- **Changing Data Types**
  1. `df.astype(new_type)` - Changes dtype.
  2. `pd.to_datetime()` - To convert a string, number, or array-like object to a datetime object.

```
df2["income"] = df2["income"].astype(int)       # change dtype

date_str = "2025-12-01"
date = pd.to_datetime(date_str)
print(date)
print(type(date))
```

- **String Cleaning**
  1. `.str.lower()` - Convert to lowercase
  2. `.str.upper()` - Convert to uppercase
  3. `.str.capitalize()` - Capitalize strings
  4. `.str.strip()` - Remove leading/trailing spaces
  5. `.str.split(" ")` - Split into parts based on a separator
  6. `.str.contains()` - Check if a value exists in string or not

```
df2["gender"].str.lower()                    # lower case
df2["gender"].str.upper()                    # upper case
df2["gender"].str.capitalize()               # capitalize

df2["gender"].str. capitalize()              # capitalize
df2["name"].str.split(" ")                   # splits into lists
type(df2["name"].str.split(" ")[0])

df2["country"].str.contains("US")
df2["country"].str.contains("india", case=False)
```

Transforming Data

- **Applying Functions**

1. `apply()` - Apply a function to a Series or DataFrame.

```
df['Age_plus_10'] = df['Age'].apply(lambda x: x + 10)
```

2. `map()` - Map a function or dictionary to a Series.

```
df['City'] = df['City'].map({'Delhi':'DL', 'Mumbai':'MB'})
```

3. `assign()` - Create new columns or modify existing columns.

```
df.assign(new_income = df["income"] * 1.1)
```

4. `replace(old, new)` - Replace specific values in a Series or DataFrame.

```
df['City'] = df['City'].replace({'Delhi':'DL', 'Mumbai':'MB'})
```

- **Renaming / Reordering**
  1. `rename()`

```
df = pd.DataFrame({
    "A": [1, 2, 3],
    "B": [4, 5, 6],
    "C": [7, 8, 9]
})

# Rename columns A -> X, B -> Y
df_renamed = df.rename(columns={"A": "X", "B": "Y"})
print(df_renamed)
```

  2. Reorder columns

```
df.columns = ["X", "Y", "Z"]  # reorders all at once

# Reorder columns to C, A, B
df_reordered = df[["C", "A", "B"]]
```

     The list must include all columns we want to keep & columns not listed will be dropped.

  3. Reset Index

```
sorted_df2.reset_index()
sorted_df2.reset_index(drop=True)   # to drop original index vals
```

- **Sorting / Ranking**
  1. `sort_values()` - Sort values in ascending or descending order.
  2. `sort_index()` - Sort indexes in ascending or descending order.
  3. `rank()` - Sort values in ascending or descending order, `method` resolves the ties.

```
# Sorting - values & index
df2.sort_values("Income")                  # sort values in ascending
df2.sort_values("Income", ascending=False) # sort values in descending
df2.sort_values(["Age", "Income"]) # sorts age,if age same then sorts income

sorted_df2 = df2.sort_values(["Age", "Income"])
sorted_df2.sort_index()

# Ranking
df2["Ranking"] = df2["Income"].rank(ascending=False, method="dense")
df2["Ranking"] = df2["Income"].rank(ascending=False, method="min")
df2["Ranking"] = df2["Income"].rank(ascending=False, method="max")
```

Grouping & Aggregation Methods

- `df.groupby()`

  We use the `groupby` method to split data into multiple groups so that we can apply some aggregation functions on it.

  ```
  df2.groupby("country")["income"].mean()    # mean income for each country
  ```

- Aggregations
  - `df.sum()`
  - `df.mean()`
  - `df.count()`
  - `df.max()`
  - `df.min()`
  - `df.std()`

  ```
  df2.groupby("gender")["income"].mean()  # mean income for each gender
  df2.groupby("country")["gender"].count() # count of gender for each country
  df2.groupby("gender")["income"].max()   # max income for each gender
  ```

- `df.aggr()`

  We use the `aggr` or `aggregation` function for multiple aggregations i.e. on multiple columns.

```python
# applies multiple aggregate functions
df.groupby("country")["income"].agg(["mean", "min", "max"])
df.groupby("country")["income"].aggregate (["mean", "min", "max"]) # alias

# rename aggregate
df.groupby("country")["income"].agg(avg_salary="mean", max_salary="max")

df.groupby("country").agg({
    "age": "mean",
    "income": "mean"
}) # aggregate on multiple cols

df.groupby("country").agg(
    avg_age=("age", "mean"),
    avg_salary= ("income", "mean")
) # rename aggregates on multiple cols
```

Reshaping Methods

We have two important methods to reshape our data - Melt & Pivot.

- **Melt (wide → long)**
  - Convert a wide format DataFrame into a long/tidy format.
  - Each variable column becomes a **row**, making it easier for plotting or analysis.
  - Syntax: `pd.melt(id_vars, value_vars, var_name, value_name)`
    1. `id_vars` - columns to keep as identifiers (don't melt).
    2. `value_vars` - columns to unpivot (melt).
    3. `var_name` - new column name for variable names.
    4. `value_name` - new column name for values.

```python
df = pd.DataFrame({
    "country": ["USA", "USA", "India", "India"],
    "year": [2020, 2021, 2020, 2021],
    "sales": [100, 120, 90, 110],
    "profit": [20, 25, 18, 22]
})

melted = df.melt(
    id_vars=["country", "year"],     # columns to keep
    value_vars=["sales", "profit"],  # columns to unpivot
    var_name="metric",               # new column name for variable
    value_name="value"     # new column name for value (default is value)
)

print(melted)
```

- **Pivot (long → wide)**
  - Convert a **long format** DataFrame back into **wide format**.
  - Syntax: `df.pivot(index, columns, values)`
    1. `index` - column to use as row index.
    2. `columns` - column to spread out as new columns.
    3. `values` - column to fill values into the new columns.

```
original = melted.pivot(
    index=["country", "year"],# cols that become the NEW row index
    columns="metric",         # col whose unique values will be the cols
    values="value (in Lakhs)" # col whose values will fill the new df
)

print(pivoted)
```

Combining & Joining Methods

- `df.merge()`

  Merge is used for SQL-like joins & combines two DataFrames based on **common columns (keys)**.

  It supports –

  - Inner join - Only keep common values of both df
  - Outer join - Keep all values of both df & replace missing with NaN
  - Left join - Keep all values of left df & replace missing with NaN
  - Right join - Keep all values of right df & replace missing with NaN

```
# Merging & Joining
df_customers = pd.DataFrame({
    "customer_id": [1, 2, 3, 4],
    "name": ["Adam", "Bob", "Charlie", "Dave"]
})

df_orders = pd.DataFrame({
    "order_id": [101, 102, 103, 104],
    "customer_id": [2, 1, 4, 5],
    "amount": [250, 120, 300, 180]
})

pd.merge(df_customers, df_orders, on="customer_id")              # Inner Join
pd.merge(df_customers, df_orders, on="customer_id", how="left")  # Left Join
pd.merge(df_customers, df_orders, on="customer_id", how="right") # Right Join
pd.merge(df_customers, df_orders, on="customer_id", how="outer") # Outer Join
```

	order_id	customer_id	amount
0	101	2	250
1	102	1	120
2	103	4	300
3	104	5	180

df_orders

	customer_id	name
0	1	Adam
1	2	Bob
2	3	Charlie
3	4	Dave

df_customers

Outer Join

	customer_id	name	order_id	amount
0	1	Adam	102.0	120.0
1	2	Bob	101.0	250.0
2	3	Charlie	Nan	Nan
3	4	Dave	103.0	300.0
4	5	Nan	104.0	180.0

We also have `Join()` method which is essentially a shortcut for merge when using indices. It is used to combine two DataFrames based on index by default & can also join on a key column.

- `df.concat()`

  We use this method for data concatenation i.e. to stack DataFrames **vertically (on top) or horizontally (side by side)**.

```python
df1 = pd.DataFrame({
    "id": [1, 2, 3],
    "name": ["Adam", "Bob", "Charlie"]
})

df2 = pd.DataFrame({
    "id": [4, 5, 6],
    "name": ["David", "Eva", "Frank"]
})

pd.concat([df1, df2])                       # Row wise concatenation
pd.concat([df1, df2], ignore_index=True) # Row wise concatenation - new index
```

	id	name
0	1	Adam
1	2	Bob
2	3	Charlie
3	4	Dave
4	5	Eva
5	6	Frank

```python
pd.concat([df1, df2], axis=1)       # Col wise concatenation
```

	id	name	id	name
0	1	Adam	4	David
1	2	Bob	5	Eva
2	3	Charlie	6	Frank

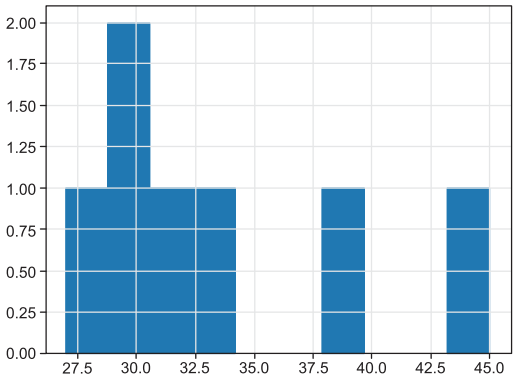
Basic Visualization

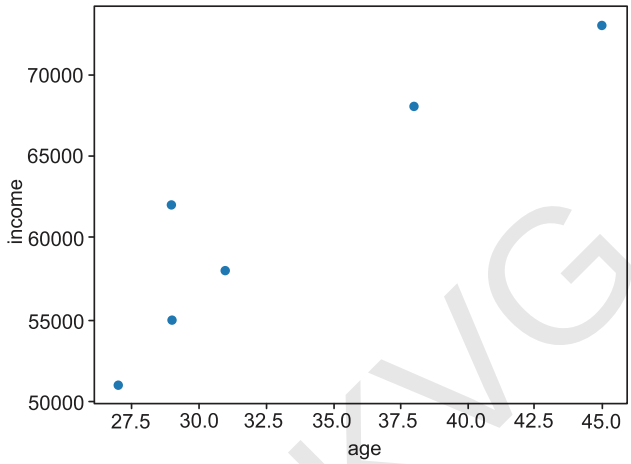
Although for most of the data visualization, we are going to use other packages like matplotlib but basic visualization can be done with Pandas using methods like `hist()` & `plot()` .

```python
# Basic Visualization

df["age"].hist()
df.plot(kind='scatter', x='age', y='income')
```

This is what the output looks like:





Keep Learning & Keep Exploring!