Pandas Series

A **Series** is a one-dimensional labeled array capable of holding any data type (integer, string, float, etc.). The labels are also called the **index**.

Key Characteristics of a Series:

- It's like a column in a table.
- It can hold a single data type, such as integers, floats, or strings.

Syntax:

```
import pandas as pd

# Creating a Series
data = [10, 20, 30, 40]
series = pd.Series(data)

# Display the series
print(series)
```

```
0    10
1    20
2    30
3    40
dtype: int64
```

Additional Example:

Creating a Series with custom index values

```
data = [100, 200, 300]
index = ['a', 'b', 'c']
series = pd.Series(data, index=index)

print(series)
```

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

Common Functions:

- **head()**: Returns the first few entries of the Series.
- **tail()**: Returns the last few entries.
- **mean()**: Calculates the mean of the Series.
- **sum()**: Returns the sum of all elements.

Example:

```
# Series operations
print(series.mean())  # Mean of the values
print(series.sum())   # Sum of the values
```

```
200.0
600
```

Pandas DataFrame

A **DataFrame** is a two-dimensional labeled data structure with columns of potentially different types. It can be thought of as a table where each column can have different types of data (e.g., integers, strings, floats, etc.).

Key Characteristics of a DataFrame:

- It is similar to an Excel spreadsheet or a SQL table.
- It can store multiple data types in different columns.

```
# Creating a DataFrame
data = {'Name': ['Alice', 'Bob', 'Charlie'], 'Age': [25, 30, 35], 'Salary': [50000,
df = pd.DataFrame(data)

# Display the DataFrame
print(df)
```

```
      Name  Age  Salary
0    Alice   25   50000
1      Bob   30   60000
2  Charlie   35   70000
```

DataFrame Constructor Parameters:

- **data**: Data to be stored in DataFrame, can be lists, dictionaries, or other data structures.
- **columns**: Labels for the columns.
- **index**: Custom index labels for the rows.

Example with Custom Columns and Index:

```
# Customizing columns and index
df_custom = pd.DataFrame(data, columns=['Name', 'Salary', 'Age'], index=['Row1', 'Ro
print(df_custom)
```

```
         Name  Salary  Age
Row1    Alice   50000   25
Row2      Bob   60000   30
Row3  Charlie   70000   35
```

Common Functions:

- **head(n): Returns the first n rows.**
- **tail(n): Returns the last n rows.**
- **describe(): Generates descriptive statistics for numerical columns.**
- **info(): Provides a concise summary of the DataFrame.**

Example:

```
# Customizing columns and index
df_custom = pd.DataFrame(data, columns=['Name', 'Salary', 'Age'], index=['Row1', 'Ro
print(df_custom)
```

```
         Name  Salary  Age
Row1    Alice   50000   25
Row2      Bob   60000   30
Row3  Charlie   70000   35
```

```
50%    30.0  60000.0
75%    32.5  65000.0
max    35.0  70000.0
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 3 entries, 0 to 2
Data columns (total 3 columns):
 #   Column   Non-Null Count  Dtype
---  ------   --------------  -----
 0   Name     3 non-null      object
 1   Age      3 non-null      int64
 2   Salary   3 non-null      int64
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 3 entries, 0 to 2
Data columns (total 3 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   Name    3 non-null      object
 1   Age     3 non-null      int64
 2   Salary  3 non-null      int64
dtypes: int64(2), object(1)
memory usage: 204.0+ bytes
None
```

Additional Key Operations:

- **Adding a new column**

```
df['Department'] = ['HR', 'Engineering', 'Marketing']
print(df)
```

```
      Name  Age  Salary   Department
0    Alice   25   50000           HR
1      Bob   30   60000  Engineering
2  Charlie   35   70000    Marketing
```

Filtering rows:

```
# Filter rows where Age > 25
filtered_df = df[df['Age'] > 25]
print(filtered_df)
```

```
      Name  Age  Salary   Department
1      Bob   30   60000  Engineering
2  Charlie   35   70000    Marketing
```

Data Input/Output: Reading Data from Files

In data analysis, it's essential to load datasets from various file formats into a structured format, such as a DataFrame, for further manipulation and analysis. **Pandas** provides convenient functions to read data from different file types like CSV, Excel, JSON, and SQL. Let's break down the commands:

1. Reading CSV Files

CSV (Comma-Separated Values) is one of the most common file formats for storing tabular data. Pandas can easily read CS files into a DataFrame using the pd.read_csv() function.

```
import pandas as pd
# Reading a CSV file
df = pd.read_csv(r"C:\Users\ASUS\Downloads\ElectricCarData_Clean.csv")
df
df.head()
```

Brand	Model	AccelSec	TopSpeed_KmH	Range_Km	Efficiency_WhKm	FastCharge_KmH	RapidCharge	PowerTrain	PlugType	BodyStyle	Segment	Seats	PriceEu
Tesla	Model 3 Long Range Dual Motor	4.6	233	450	161	940	Yes	AWD	Type 2 CCS	Sedan	D	5	554
Volkswagen	ID.3 Pure	10.0	160	270	167	250	Yes	RWD	Type 2 CCS	Hatchback	C	5	300
Polestar	2	4.7	210	400	181	620	Yes	AWD	Type 2 CCS	Liftback	D	5	564
BMW	iX3	6.8	180	360	206	560	Yes	RWD	Type 2 CCS	SUV	D	5	680
Honda	e	9.5	145	170	168	190	Yes	RWD	Type 2 CCS	Hatchback	B	4	329

2. Reading Excel Files

Pandas can also handle Excel files with multiple sheets. The pd.read_excel() function can be used to read an Excel file into a DataFrame.

```
# Reading an Excel file
df = pd.read_excel('file.xlsx')
# Displaying the first few rows
print(df.head())
```

3. Reading JSON Files

JSON (JavaScript Object Notation) is a popular format for exchanging data, especially from web APIs. Pandas allows easy conversion of JSON files into a DataFrame with the pd.read_json() function.

```
# Reading a JSON file
df = pd.read_json('file.json')
# Displaying the first few rows
print(df.head())
```

JSON structure: Pandas expects JSON to be structured as an array of objects (dictionaries). If the file format is complex, you might need to normalize the data.

4. Reading SQL Queries

Pandas integrates with SQL databases to run SQL queries and return the result as a DataFrame. The pd.read_sql() function requires a SQL query and a database connection.

```
import sqlite3
# Establishing a connection to the database
connection = sqlite3.connect('database.db')
# Reading data from SQL using a query
df = pd.read_sql('SELECT * FROM table_name', connection)
# Displaying the first few rows
print(df.head())
```

connection: A connection object that represents the database.

query: The SQL query you want to execute, such as SELECT * FROM table_name.

Data Input/Output: Writing Data to Files

After performing data analysis or transformations, saving the processed data into different file formats is often necessary. **Pandas** provides methods to export DataFrames into various file formats such as CSV, Excel, JSON, and SQL databases. Below are the main functions for writing data to files and an explanation of each.

1. Writing to CSV Files

The to_csv() function allows you to save a Pandas DataFrame as a CSV (Comma-Separated Values) file. This is one of the most common formats for exporting data.

```
# Writing the DataFrame to a CSV file
df.to_csv('file.csv', index=False)
# index=False: Prevents Pandas from writing row indices to the file.
```

2. Writing to Excel Files

The to_excel() function saves a DataFrame to an Excel file. Pandas can export to multiple sheets if needed.

```
# Writing the DataFrame to an Excel file
df.to_excel('file.xlsx', index=False)
```

3. Writing to JSON Files

The to_json() function allows exporting the DataFrame as a JSON (JavaScript Object Notation) file, which is widely used in web development and APIs.

```python
# Writing the DataFrame to a JSON file
df.to_json('file.json', orient='records')
# orient='records': Each row is written as a JSON object, with column names as keys.
```

orient='records': This structure is useful when you want each row in the DataFrame to be represented as a JSON object. Other orientations like 'columns' and 'index' are available based on your use case.

4. Writing to SQL Databases

Pandas can write DataFrames directly to an SQL database using the to_sql() function. This requires an active database connection and a table name.

```python
# Writing the DataFrame to a SQL table
df.to_sql('table_name', connection, if_exists='replace', index=False)
# connection: The active database connection object.
# if_exists='replace': If the table already exists, it will be replaced.
```

Data Inspection and Understanding

Before performing data analysis or transformations, it's important to inspect the dataset and understand its structure and content. **Pandas** offers several methods to explore a DataFrame, including viewing rows, checking the summary of the dataset, and obtaining statistical details. Below are key functions for data inspection and an explanation of each.

1. df.head(n) – Display First n Rows

The head() function allows you to preview the first few rows of the DataFrame. This is useful for getting a quick glimpse of the data.

n: The number of rows you want to display. If no value is provided, it defaults to 5.

```python
# Displaying the first 5 rows (default)
df.head()
# Displaying the first n rows
df.head(n=4)
```

	Brand	Model	AccelSec	TopSpeed_KmH	Range_Km	Efficiency_WhKm	FastCharge_KmH	RapidCharge	PowerTrain	PlugType	BodyStyle	Segment	Seats	Price
0	Tesla	Model 3 Long Range Dual Motor	4.6	233	450	161	940	Yes	AWD	Type 2 CCS	Sedan	D	5	[illegible]
1	Volkswagen	ID.3 Pure	10.0	160	270	167	250	Yes	RWD	Type 2 CCS	Hatchback	C	5	[illegible]
2	Polestar	2	4.7	210	400	181	620	Yes	AWD	Type 2 CCS	Liftback	D	5	[illegible]
3	BMW	iX3	6.8	180	360	206	560	Yes	RWD	Type 2 CCS	SUV	D	5	[illegible]

2. df.tail(n) – Display Last n Rows

The tail() function works similarly to head(), but it shows the last few rows of the DataFrame. This is often useful for examining data at the end of a dataset.

n: The number of rows to display from the bottom of the DataFrame.

```
# Displaying the last 5 rows (default)
df.tail()
# Displaying the last n rows
df.tail(n=4)
```

	Brand	Model	AccelSec	TopSpeed_KmH	Range_Km	Efficiency_WhKm	FastCharge_KmH	RapidCharge	PowerTrain	PlugType	BodyStyle	Segment	Seats	P
99	Audi	e-tron S Sportback 55 quattro	4.5	210	335	258	540	Yes	AWD	Type 2 CCS	SUV	E	5	
100	Nissan	Ariya e-4ORCE 63kWh	5.9	200	325	194	440	Yes	AWD	Type 2 CCS	Hatchback	C	5	
101	Nissan	Ariya e-4ORCE 87kWh Performance	5.1	200	375	232	450	Yes	AWD	Type 2 CCS	Hatchback	C	5	
102	Byton	M-Byte 95 kWh 2WD	7.5	190	400	238	480	Yes	AWD	Type 2 CCS	SUV	E	5	

3. df.info() – Summary of the DataFrame

The info() function provides a concise summary of the DataFrame, including the index, column names, data types, and non-null counts for each column. It is an essential method to quickly assess the structure of the dataset.

```
# Get a summary of the DataFrame
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 103 entries, 0 to 102
Data columns (total 14 columns):
 #   Column           Non-Null Count  Dtype
---  ------           --------------  -----
 0   Brand            103 non-null    object
 1   Model            103 non-null    object
 2   AccelSec         103 non-null    float64
 3   TopSpeed_KmH     103 non-null    int64
 4   Range_Km         103 non-null    int64
 5   Efficiency_WhKm  103 non-null    int64
 6   FastCharge_KmH   103 non-null    object
 7   RapidCharge      103 non-null    object
 8   PowerTrain       103 non-null    object
 9   PlugType         103 non-null    object
 10  BodyStyle        103 non-null    object
 11  Segment          103 non-null    object
 12  Seats            103 non-null    int64
 13  PriceEuro        103 non-null    int64
dtypes: float64(1), int64(5), object(8)
memory usage: 11.4+ KB
```

This method helps identify missing values and data types, making it useful when preparing the data for analysis.

4. df.describe() – Descriptive Statistics for Numerical Columns

The describe() function generates descriptive statistics of numerical columns, such as count, mean, standard deviation, minimum, and maximum values. This helps in understanding the distribution of the data.

```
# Get descriptive statistics
df.describe()
```

	AccelSec	TopSpeed_KmH	Range_Km	Efficiency_WhKm	Seats	PriceEuro
count	103.000000	103.000000	103.000000	103.000000	103.000000	103.000000
mean	7.396117	179.194175	338.786408	189.165049	4.883495	55811.563107
std	3.017430	43.573030	126.014444	29.566839	0.795834	34134.665280
min	2.100000	123.000000	95.000000	104.000000	2.000000	20129.000000
25%	5.100000	150.000000	250.000000	168.000000	5.000000	34429.500000
50%	7.300000	160.000000	340.000000	180.000000	5.000000	45000.000000
75%	9.000000	200.000000	400.000000	203.000000	5.000000	65000.000000
max	22.400000	410.000000	970.000000	273.000000	7.000000	215000.000000

For numerical columns: It displays the count, mean, standard deviation (std), min, 25th percentile (25%), 50th percentile (median or 50%), 75th percentile (75%), and max values.

the parameter include='all' can be passed to get statistics for categorical columns as well

5. df.shape – Dimensions of the DataFrame

The shape attribute returns a tuple representing the dimensions of the DataFrame, i.e., the number of rows and columns.

```
# Get the dimensions of the DataFrame
df.shape
```

```
(103, 14)
```

Output is in the form of (rows, columns), helping you quickly understand the size of your dataset.

6. df.columns – List Column Names

The columns attribute provides a list of the column names in the DataFrame. This is particularly helpful for knowing which columns you can access for analysis.

```
# Get the column names
df.columns
```

```
Index(['Brand', 'Model', 'AccelSec', 'TopSpeed_KmH', 'Range_Km',
       'Efficiency_WhKm', 'FastCharge_KmH', 'RapidCharge', 'PowerTrain',
       'PlugType', 'BodyStyle', 'Segment', 'Seats', 'PriceEuro'],
      dtype='object')
```

Output is a list-like object with the names of the columns in the DataFrame.

7. df.index – Access the Index (Row Labels)

The index attribute returns the index (row labels) of the DataFrame. By default, these are integers, but the index can also contain meaningful labels, like dates or IDs

```python
# Get the row labels (index)
df.index
```

```
RangeIndex(start=0, stop=103, step=1)
```

Output is an Index object that shows how rows are labeled. This can be useful when working with time series or datasets where the row index holds important information.

Data Types: Inspecting and Converting Data Types

Understanding the data types of each column in a DataFrame is critical for proper data analysis. Pandas provides methods to both inspect and change the data types of columns, which is useful when dealing with numerical operations, categorizations, or formatting issues. Below are key functions for working with data types in Pandas and explanations for their usage.

1. df.dtypes – Display Data Types of Each Column

The dtypes attribute returns the data type of each column in the DataFrame. This helps in determining whether a column is numeric, categorical, or textual, allowing you to apply the correct operations or transformations.

```python
# Display the data types of each column
df.dtypes
```

```
Brand               object
Model               object
AccelSec           float64
TopSpeed_KmH         int64
Range_Km             int64
Efficiency_WhKm      int64
FastCharge_KmH      object
RapidCharge         object
PowerTrain          object
PlugType            object
BodyStyle           object
Segment             object
Seats                int64
PriceEuro            int64
dtype: object
```

- **Output**: The result is a series where each column name is associated with its respective data type, such as int64, float64, object (for strings or mixed types), or datetime64.

2. df.astype(type) – Convert the Data Type of a Column

The astype() function is used to convert the data type of one or more columns. This is useful when, for example, you need to change a numerical column from float to integer, convert strings to dates, or ensure categorical data is handled properly.

```
# Converting a column to a different data type
df['Range_Km'] = df['Range_Km'].astype(float)
print(df.head(4))
```

```
        Brand                          Model  AccelSec  TopSpeed_KmH  \
0       Tesla  Model 3 Long Range Dual Motor       4.6           233
1  Volkswagen                      ID.3 Pure      10.0           160
2    Polestar                              2       4.7           210
3         BMW                           iX3        6.8           180

   Range_Km  Efficiency_WhKm FastCharge_KmH RapidCharge PowerTrain  \
0     450.0              161            940         Yes        AWD
1     270.0              167            250         Yes        RWD
2     400.0              181            620         Yes        AWD
3     360.0              206            560         Yes        RWD

     PlugType  BodyStyle Segment  Seats  PriceEuro
0  Type 2 CCS      Sedan       D      5      55480
1  Type 2 CCS  Hatchback       C      5      30000
2  Type 2 CCS   Liftback       D      5      56440
3  Type 2 CCS        SUV       D      5      68040
```

type: The desired data type for the conversion, such as int, float, str, category, or datetime64.

Data Selection and Indexing

Selecting and accessing specific parts of a DataFrame is a fundamental aspect of data manipulation in Pandas. Whether you want to access columns, select specific rows, or filter data based on conditions, Pandas provides several powerful methods to handle this efficiently. Below are explanations and examples of how to select and index data in Pandas.

1. Selecting Columns

There are two main ways to select columns in a DataFrame: accessing a single column and accessing multiple columns.

df['column_name'] – Access a Single Column

You can select a single column from a DataFrame using the bracket notation ['column_name']. This will return a Series.

```
# Accessing a single column
df['Brand']
```

```
0           Tesla
1      Volkswagen
2        Polestar
3             BMW
4           Honda
          ...
98         Nissan
99           Audi
100        Nissan
101        Nissan
102         Byton
Name: Brand, Length: 103, dtype: object
```

- **Output**: Returns a Pandas Series with data from the specified column.

df[['col1', 'col2']] – Access Multiple Columns

You can select multiple columns by passing a list of column names inside double brackets [['col1', 'col2']].

```python
# Accessing multiple columns
df[['Brand','Model']]
```

	Brand	Model
0	Tesla	Model 3 Long Range Dual Motor
1	Volkswagen	ID.3 Pure
2	Polestar	2
3	BMW	iX3
4	Honda	e
...	...	...
98	Nissan	Ariya 63kWh
99	Audi	e-tron S Sportback 55 quattro
100	Nissan	Ariya e-4ORCE 63kWh
101	Nissan	Ariya e-4ORCE 87kWh Performance
102	Byton	M-Byte 95 kWh 2WD

103 rows × 2 columns

Output: Returns a DataFrame containing only the specified columns

2. Selecting Rows

Pandas provides two main methods for selecting rows: loc[] for label-based selection and iloc[] for position-based selection.

df.loc[] – Label-Based Selection

The loc[] method is used for label-based selection of rows and columns. You can specify row labels (or indices) and column names.

```python
# Selecting a row by label (index 1) and accessing the 'Name' column
row_label = df.loc[1, 'Brand']
print(row_label)
```

```
Volkswagen
```

df.iloc[] – Integer-Location-Based Selection

The iloc[] method is used for selection based on the integer position of rows and columns.

```python
# Selecting the second row (index 1) and accessing the 'Name' column
row_pos = df.iloc[1, 0]
print(row_pos)
```

```
Volkswagen
```

Selecting Rows by Label or Position:

3. Fast Access to Scalar Values

For fast access to individual scalar values, Pandas offers at[] and iat[].

df.at[] – Fast Label-Based Access

Use at[] for fast access to a single scalar value based on row and column labels.

```
# Accessing the value in row 1, 'Name' column
value_at = df.at[1, 'Brand']
print(value_at)
```

```
Volkswagen
```

df.iat[] – Fast Integer-Location-Based Access

```
# Accessing the value at row index 1 and column index 0
value_iat = df.iat[1, 0]
print(value_iat)
```

```
Volkswagen
```

4. Filtering Rows

You can filter rows in a DataFrame based on conditions. The condition returns a boolean array, which is used to filter the DataFrame.

df[df['column'] > value] – Filter Rows Based on Conditions

```
# Filtering rows where 'Age' is greater than 30
filtered_df = df[df['Range_Km'] > 530]
print(filtered_df)
```

```
        Brand                  Model  AccelSec  TopSpeed_KmH  Range_Km  \
5       Lucid                    Air       2.8           250     610.0
33      Tesla  Cybertruck Tri Motor        3.0           210     750.0
48  Lightyear                    One      10.0           150     575.0
51      Tesla               Roadster       2.1           410     970.0

    Efficiency_WhKm FastCharge_KmH RapidCharge PowerTrain    PlugType  \
5               180            620         Yes        AWD  Type 2 CCS
33              267            710         Yes        AWD  Type 2 CCS
48              104            540         Yes        AWD  Type 2 CCS
51              206            920         Yes        AWD  Type 2 CCS

   BodyStyle Segment  Seats  PriceEuro
5      Sedan       F      5     105000
33    Pickup       N      6      75000
48  Liftback       F      5     149000
51    Cabrio       S      4     215000
```

Data Cleaning

Data cleaning is an essential step in preparing your dataset for analysis. It involves handling missing values, removing duplicates, and replacing unwanted values. Pandas offers several methods to perform these tasks efficiently. Below are the key functions for data cleaning, along with explanations and examples.

```
import pandas as pd

df=pd.read_csv(r"/complete_employee_dataset.csv")
```

1. Handling Missing Data

Missing data is common in real-world datasets, and Pandas provides various functions to detect, remove, or fill in missing values.

df.isnull() – Detect Missing Values

The isnull() function detects missing values in the DataFrame. It returns a DataFrame of the same shape with True for missing values (NaN) and False for non-missing values.

```
# Detect missing values
df.isnull()
```

```
df.isnull()
SS
```

me	Age	Salary	Department	Hire_Date	Experience_Years	Bonus	Promotion_Eligibility	Projects_Completed	Working_Remote
se	False	False	False	False	False	False	False	False	False
se	False	False	False	False	False	False	False	False	False
se	False	False	False	False	False	False	False	False	False
se	False	False	False	False	False	False	False	False	False
se	False	False	False	False	False	False	False	False	False
se	False	False	False	False	False	False	False	False	False
se	False	False	False	False	False	False	False	False	False
se	False	False	False	False	False	False	False	False	False

df.notnull() – Detect Non-Missing Values

The notnull() function works similarly to isnull(), but it returns True for non-missing values and False for missing ones.

```
# Detect non-missing values
df.notnull()
```

df.dropna() – Drop Rows or Columns with Missing Values

The dropna() function allows you to remove rows or columns that contain missing values (NaN). By default, it drops rows with any missing values.

```
# Drop rows with missing values
df.dropna()
```

```
df.notnull()
```

	Name	Age	Salary	Department	Hire_Date	Experience_Years	Bonus	Promotion_Eligibility	Projects_Completed	Working_Remote
0	True	True	True	True	True	True	True	True	True	True
1	True	True	True	True	True	True	True	True	True	True
2	True	True	True	True	True	True	True	True	True	True
3	True	True	True	True	True	True	True	True	True	True
4	True	True	True	True	True	True	True	True	True	True
5	True	True	True	True	True	True	True	True	True	True
6	True	True	True	True	True	True	True	True	True	True
7	True	True	True	True	True	True	True	True	True	True

- **axis=0**: Drop rows (default).
- **axis=1**: Drop columns.

`df.fillna(value)` – Fill Missing Values

The `fillna()` function allows you to replace missing values with a specified value. You can use a constant or a more complex filling method, like forward-fill (`ffill`) or backward-fill (`bfill`).

```
# Fill missing values with a specific value
df.fillna(value)
```

```
df.fillna(0)
```

	Name	Age	Salary	Department	Hire_Date	Experience_Years	Bonus	Promotion_Eligibility	Projects_Completed	Working_Remote
0	John	25	50000	HR	2020-01-15	3	5000	Yes	5	No
1	Jane	32	60000	Finance	2018-07-23	8	6000	No	12	Yes
2	Doe	45	70000	Engineering	2016-03-12	10	7000	Yes	15	No
3	Alice	28	48000	Marketing	2019-11-01	4	4800	Yes	8	Yes
4	Bob	23	51000	Sales	2021-06-15	2	5100	No	3	No
5	Eve	35	62000	IT	2017-09-30	5	6200	Yes	10	Yes
6	Mark	40	73000	Finance	2015-05-11	12	7300	No	20	Yes
7	Tom	29	55000	Engineering	2022-02-18	1	5500	Yes	4	No

Removing Duplicates

Duplicate data can skew the results of your analysis, so it's important to identify and remove them.

df.drop_duplicates() – Remove Duplicate Rows

The drop_duplicates() function removes duplicate rows from the DataFrame. By default, it keeps the first occurrence of a duplicate row and removes subsequent duplicates

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

```
df.drop_duplicates()
```

	Name	Age	Salary	Department	Hire_Date	Experience_Years	Bonus	Promotion_Eligibility	Projects_Completed	Working_Remote
0	John	25	50000	HR	2020-01-15	3	5000	Yes	5	No
1	Jane	32	60000	Finance	2018-07-23	8	6000	No	12	Yes
2	Doe	45	70000	Engineering	2016-03-12	10	7000	Yes	15	No
3	Alice	28	48000	Marketing	2019-11-01	4	4800	Yes	8	Yes
4	Bob	23	51000	Sales	2021-06-15	2	5100	No	3	No
5	Eve	35	62000	IT	2017-09-30	5	6200	Yes	10	Yes
6	Mark	40	73000	Finance	2015-05-11	12	7300	No	20	Yes
7	Tom	29	55000	Engineering	2022-02-18	1	5500	Yes	4	No

Replacing Values

Sometimes, you may want to replace specific values in your DataFrame, such as correcting errors or standardizing formats.

df.replace(to_replace, value) – Replace Specific Values

The replace() function allows you to replace specific values in the DataFrame. You can replace a single value, multiple values, or use more complex mappings.

```
# Replace specific values in the DataFrame
df.replace(to_replace, value)
```

```
df.replace({'No': 'Not Eligible', 'Yes': 'Eligible'})
```

ame	Age	Salary	Department	Hire_Date	Experience_Years	Bonus	Promotion_Eligibility	Projects_Completed	Working_Remote
ohn	25	50000	HR	2020-01-15	3	5000	Eligible	5	Not Eligible
ane	32	60000	Finance	2018-07-23	8	6000	Not Eligible	12	Eligible
Doe	45	70000	Engineering	2016-03-12	10	7000	Eligible	15	Not Eligible
lice	28	48000	Marketing	2019-11-01	4	4800	Eligible	8	Eligible
Bob	23	51000	Sales	2021-06-15	2	5100	Not Eligible	3	Not Eligible
Eve	35	62000	IT	2017-09-30	5	6200	Eligible	10	Eligible
ark	40	73000	Finance	2015-05-11	12	7300	Not Eligible	20	Eligible
Tom	29	55000	Engineering	2022-02-18	1	5500	Eligible	4	Not Eligible

Data Manipulation

Data manipulation involves changing, updating, and reorganizing the structure of your DataFrame. Pandas provides various functions to rename columns, sort data, add or modify columns, and drop unnecessary rows or columns. Below are the key functions for data manipulation with explanations and examples.

1. Renaming Columns/Index

Renaming columns or the index in a DataFrame is often required when the column names are not descriptive or need to be standardized for analysis.

df.rename(columns={'old_name': 'new_name'}) – Renaming Columns

The rename() function allows you to rename columns in a DataFrame by passing a dictionary that maps old column names to new ones.

```
# Rename columns in the DataFrame
df.rename(columns={'old_name': 'new_name'})
```

```
df.rename(columns={'Department': 'Dept'}, inplace=True)
df
```

	Name	Age	Salary	Dept	Hire_Date	Experience_Years	Bonus	Promotion_Eligibility	Projects_Completed	Working_Remote
0	John	25	50000	HR	2020-01-15	3	5000	Yes	5	No
1	Jane	32	60000	Finance	2018-07-23	8	6000	No	12	Yes
2	Doe	45	70000	Engineering	2016-03-12	10	7000	Yes	15	No
3	Alice	28	48000	Marketing	2019-11-01	4	4800	Yes	8	Yes
4	Bob	23	51000	Sales	2021-06-15	2	5100	No	3	No
5	Eve	35	62000	IT	2017-09-30	5	6200	Yes	10	Yes
6	Mark	40	73000	Finance	2015-05-11	12	7300	No	20	Yes
7	Tom	29	55000	Engineering	2022-02-18	1	5500	Yes	4	No

df.set_index('column_name') – Set a Column as the Index

The set_index() function sets a specific column as the DataFrame's index, replacing the default numeric index.

```
# Set a column as the index
df.set_index('column_name')
```

```
df.set_index('Name', inplace=True)
df
```

	Age	Salary	Dept	Hire_Date	Experience_Years	Bonus	Promotion_Eligibility	Projects_Completed	Working_Remote
Name									
John	25	50000	HR	2020-01-15	3	5000	Yes	5	No
Jane	32	60000	Finance	2018-07-23	8	6000	No	12	Yes
Doe	45	70000	Engineering	2016-03-12	10	7000	Yes	15	No
Alice	28	48000	Marketing	2019-11-01	4	4800	Yes	8	Yes
Bob	23	51000	Sales	2021-06-15	2	5100	No	3	No
Eve	35	62000	IT	2017-09-30	5	6200	Yes	10	Yes
Mark	40	73000	Finance	2015-05-11	12	7300	No	20	Yes
Tom	29	55000	Engineering	2022-02-18	1	5500	Yes	4	No

Sorting Data

Sorting is useful for organizing your data by specific columns or by the index.

df.sort_values(by='column_name') – Sort by Specific Column(s)

The sort_values() function sorts the DataFrame by the specified column(s). You can sort in ascending (default) or descending order

```
# Sort DataFrame by a specific column
df.sort_values(by='column_name')
```

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

	Age	Salary	Dept	Hire_Date	Experience_Years	Bonus	Promotion_Eligibility	Projects_Completed	Working_Remote
Name									
Mark	40	73000	Finance	2015-05-11	12	7300	No	20	Yes
Doe	45	70000	Engineering	2016-03-12	10	7000	Yes	15	No
Eve	35	62000	IT	2017-09-30	5	6200	Yes	10	Yes
Jane	32	60000	Finance	2018-07-23	8	6000	No	12	Yes
Tom	29	55000	Engineering	2022-02-18	1	5500	Yes	4	No
Bob	23	51000	Sales	2021-06-15	2	5100	No	3	No
John	25	50000	HR	2020-01-15	3	5000	Yes	5	No
Alice	28	48000	Marketing	2019-11-01	4	4800	Yes	8	Yes

df.sort_index() – Sort by Index

The sort_index() function sorts the DataFrame based on the index values. This is useful when you want to reorder rows by their index labels.

```python
# Sort DataFrame by index
df.sort_index()
```

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

Name	Age	Salary	Dept	Hire_Date	Experience_Years	Bonus	Promotion_Eligibility	Projects_Completed	Working_Remote
Alice	28	48000	Marketing	2019-11-01	4	4800	Yes	8	Yes
Bob	23	51000	Sales	2021-06-15	2	5100	No	3	No
Doe	45	70000	Engineering	2016-03-12	10	7000	Yes	15	No
Eve	35	62000	IT	2017-09-30	5	6200	Yes	10	Yes
Jane	32	60000	Finance	2018-07-23	8	6000	No	12	Yes
John	25	50000	HR	2020-01-15	3	5000	Yes	5	No
Mark	40	73000	Finance	2015-05-11	12	7300	No	20	Yes
Tom	29	55000	Engineering	2022-02-18	1	5500	Yes	4	No

3. Adding/Modifying Columns

You can create new columns or modify existing ones by assigning values to them.

df['new_column'] = data – Create or Modify a Column

```python
# Create a new column or modify an existing column
df['new_column'] = data
```

```python
df['Bonus Percentage'] = (df['Bonus'] / df['Salary']) * 100
df
```

Name	Age	Salary	Dept	Hire_Date	Experience_Years	Bonus	Promotion_Eligibility	Projects_Completed	Working_Remote	Bonus Percentage
Alice	28	48000	Marketing	2019-11-01	4	4800	Yes	8	Yes	10.0
Bob	23	51000	Sales	2021-06-15	2	5100	No	3	No	10.0
Doe	45	70000	Engineering	2016-03-12	10	7000	Yes	15	No	10.0
Eve	35	62000	IT	2017-09-30	5	6200	Yes	10	Yes	10.0
Jane	32	60000	Finance	2018-07-23	8	6000	No	12	Yes	10.0
John	25	50000	HR	2020-01-15	3	5000	Yes	5	No	10.0
Mark	40	73000	Finance	2015-05-11	12	7300	No	20	Yes	10.0
Tom	29	55000	Engineering	2022-02-18	1	5500	Yes	4	No	10.0

df.assign(new_col=data) – Add a Column and Return a New DataFrame

The assign() function is similar to direct assignment but it returns a new DataFrame rather than modifying the original one.

```python
# Add a new column and return a new DataFrame
new_df = df.assign(new_col=data)
```

Dropping Rows/Columns

Sometimes, you need to remove unnecessary columns or rows from your DataFrame.

df.drop(['column_name'], axis=1) – Drop Columns

```python
# Drop columns from the DataFrame
df.drop(['column_name'], axis=1)
```

df.drop([row_index], axis=0) – Drop Rows

```
# Drop rows from the DataFrame
df.drop([row_index], axis=0)
```

Merging and Concatenating DataFrames

Pandas provides powerful tools to combine DataFrames either by concatenating them or by merging them based on common columns or indexes.

Concatenation

Concatenation is used to combine two or more DataFrames either row-wise or column-wise. The pd.concat() function is typically used for this purpose.

pd.concat([df1, df2], axis=0) – Concatenate DataFrames

The concat() function concatenates two or more DataFrames along a particular axis (rows or columns).

- **df1, df2**: The DataFrames to concatenate.
- **axis=0**: Concatenation along rows (default).
- **axis=1**: Concatenation along columns.

```
# Concatenate DataFrames row-wise or column-wise
pd.concat([df1, df2], axis=0)  # Concatenate row-wise (default)
pd.concat([df1, df2], axis=1)  # Concatenate column-wise
```

Concatenating Column-wise:

```
# Concatenating column-wise
df_concat_cols = pd.concat([df1, df2], axis=1)
print(df_concat_cols)
```

```
import pandas as pd

# Creating two DataFrames
data1 = {'Employee ID': [101, 102],
         'Name': ['Alice', 'Bob'],
         'Department': ['IT', 'HR'],
         'Salary': [70000, 80000]}

data2 = {'Employee ID': [103, 104],
         'Name': ['Charlie', 'David'],
         'Department': ['Finance', 'Marketing'],
         'Salary': [75000, 90000]}

df1 = pd.DataFrame(data1)
df2 = pd.DataFrame(data2)

# Concatenating row-wise (default axis=0)
concat_row = pd.concat([df1, df2], axis=0, ignore_index=True)
print("Row-wise Concatenation:")
print(concat_row)

# Concatenating column-wise (axis=1)
concat_column = pd.concat([df1, df2], axis=1)
print("\nColumn-wise Concatenation:")
print(concat_column)
```

```
Row-wise Concatenation:
   Employee ID     Name Department  Salary
0          101    Alice         IT   70000
1          102      Bob         HR   80000
2          103  Charlie    Finance   75000
3          104    David  Marketing   90000

Column-wise Concatenation:
   Employee ID   Name Department  Salary  Employee ID     Name Department  \
0          101  Alice         IT   70000          103  Charlie    Finance
1          102    Bob         HR   80000          104    David  Marketing

   Salary
0   75000
1   90000
```

Merging

Merging is used when you want to combine two DataFrames based on a common column or index. The pd.merge() function is typically used for this.

pd.merge(df1, df2, on='key') – Merge Two DataFrames on a Key

The merge() function merges two DataFrames based on a common column (key).

- **df1, df2**: The DataFrames to merge.
- **on**: The column (or columns) to merge on (common key).

```python
# Concatenating column-wise
df_concat_cols = pd.concat([df1, df2], axis=1)
print(df_concat_cols)
```

```python
# Creating two DataFrames with a common 'Employee ID' column
data3 = {'Employee ID': [101, 102, 103],
         'Name': ['Alice', 'Bob', 'Charlie'],
         'Department': ['IT', 'HR', 'Finance']}

data4 = {'Employee ID': [101, 103, 104],
         'Salary': [70000, 75000, 90000]}

df3 = pd.DataFrame(data3)
df4 = pd.DataFrame(data4)

# Merging DataFrames on 'Employee ID'
merged_df = pd.merge(df3, df4, on='Employee ID', how='inner')
print("\nMerged DataFrame (Inner Join):")
print(merged_df)

# Left join example
left_merged_df = pd.merge(df3, df4, on='Employee ID', how='left')
print("\nMerged DataFrame (Left Join):")
print(left_merged_df)

# Outer join example
outer_merged_df = pd.merge(df3, df4, on='Employee ID', how='outer')
print("\nMerged DataFrame (Outer Join):")
print(outer_merged_df)
```

```
Merged DataFrame (Inner Join):
   Employee ID     Name Department  Salary
0          101    Alice         IT   70000
1          103  Charlie    Finance   75000

Merged DataFrame (Left Join):
   Employee ID     Name Department   Salary
0          101    Alice         IT  70000.0
1          102      Bob         HR      NaN
2          103  Charlie    Finance  75000.0

Merged DataFrame (Outer Join):
   Employee ID     Name Department   Salary
0          101    Alice         IT  70000.0
1          102      Bob         HR      NaN
2          103  Charlie    Finance  75000.0
3          104      NaN        NaN  90000.0
```

Joining DataFrames

Joining is similar to merging, but it is typically used to join DataFrames by their index. The join() function allows you to combine DataFrames using their index.

df1.join(df2) – Join DataFrames by Index

The join() function joins two DataFrames based on their index.

- **df1, df2**: The DataFrames to join.

```
# Concatenating column-wise
df_concat_cols = pd.concat([df1, df2], axis=1)
print(df_concat_cols)
```

-

```
# Creating two DataFrames with index as Employee ID
data5 = {'Name': ['Alice', 'Bob', 'Charlie'],
         'Department': ['IT', 'HR', 'Finance']}

data6 = {'Salary': [70000, 80000, 75000]}

df5 = pd.DataFrame(data5, index=[101, 102, 103])
df6 = pd.DataFrame(data6, index=[101, 102, 103])

# Join DataFrames based on index
joined_df = df5.join(df6)
print("\nJoined DataFrame (on index):")
print(joined_df)
```

```
Joined DataFrame (on index):
        Name Department  Salary
101    Alice         IT   70000
102      Bob         HR   80000
103  Charlie    Finance   75000
```

Time Series and Date Handling

Working with dates and times is a crucial part of many data analysis tasks. Pandas provides powerful tools for manipulating time series data. Let's cover **datetime conversion** and **resampling** with content, functions, and examples.

Datetime Conversion

Pandas has the pd.to_datetime() function to convert a column or Series to a datetime format, making it easier to handle and manipulate date/time data.

Function:

```
pd.to_datetime(df['column'])
```

- Converts a column or Series into the datetime64 format.
- Once converted, you can perform various operations like extracting the year, month, day, or even filtering by date.

Example:

```python
import pandas as pd

# Sample DataFrame with a date column as a string
data = {'Employee ID': [101, 102, 103],
        'Name': ['Alice', 'Bob', 'Charlie'],
        'Joining Date': ['2022-01-15', '2023-03-22', '2021-11-05'],
        'Salary': [70000, 80000, 75000]}

df = pd.DataFrame(data)

# Convert 'Joining Date' to datetime format
df['Joining Date'] = pd.to_datetime(df['Joining Date'])

# Display the DataFrame with 'Joining Date' as a datetime object
print("DataFrame with Datetime Conversion:")
print(df)

# Extracting the Year, Month, and Day from the 'Joining Date'
df['Year'] = df['Joining Date'].dt.year
df['Month'] = df['Joining Date'].dt.month
df['Day'] = df['Joining Date'].dt.day

print("\nDataFrame after extracting Year, Month, and Day:")
print(df)
```

```
DataFrame with Datetime Conversion:
   Employee ID     Name Joining Date  Salary
0          101    Alice   2022-01-15   70000
1          102      Bob   2023-03-22   80000
2          103  Charlie   2021-11-05   75000

DataFrame after extracting Year, Month, and Day:
   Employee ID     Name Joining Date  Salary  Year  Month  Day
0          101    Alice   2022-01-15   70000  2022      1   15
1          102      Bob   2023-03-22   80000  2023      3   22
2          103  Charlie   2021-11-05   75000  2021     11    5
```

Output:

- The **'Joining Date'** column is now converted to the datetime64 format.
- You can extract year, month, and day using dt.year, dt.month, and dt.day.

Resampling

Resampling is a method of frequency conversion and is essential in working with time series data. You can resample time series data to different time intervals like day, month, or year, and apply aggregation functions like `mean()`, `sum()`, etc.

```
df.resample('M').mean()
```

Function:

- Resamples data by month. Other time periods include:
  - **'D'**: Daily.
  - **'W'**: Weekly.
  - **'Y'**: Yearly.
- Aggregates data, typically by taking the mean, sum, etc., over the new time interval.
- Requires a datetime index for resampling to work.

Example:

```
# Creating a DataFrame with daily sales data
date_range = pd.date_range(start='2023-01-01', end='2023-04-30', freq='D')
sales_data = {'Date': date_range, 'Sales': range(1, len(date_range) + 1)}

df_sales = pd.DataFrame(sales_data)

# Setting the 'Date' column as the index (required for resampling)
df_sales.set_index('Date', inplace=True)

print("Original Daily Sales Data:")
print(df_sales.head())

# Resampling the sales data by month and calculating the mean
monthly_sales = df_sales.resample('M').mean()

print("\nResampled Monthly Sales Data (Mean):")
print(monthly_sales)
```

```
Original Daily Sales Data:
            Sales
Date
2023-01-01      1
2023-01-02      2
2023-01-03      3
2023-01-04      4
2023-01-05      5

Resampled Monthly Sales Data (Mean):
            Sales
Date
2023-01-31   16.0
2023-02-28   45.5
2023-03-31   75.0
2023-04-30  105.5
<ipython-input-5-3169154fff80>:14: FutureWarning: 'M' is deprecated and will be removed in a future version, ple
  monthly_sales = df_sales.resample('M').mean()
```

- **Datetime Conversion (pd.to_datetime())**: Converts string dates into Pandas datetime format for easier manipulation.

- **Resampling (df.resample())**: Resamples time series data to a different frequency (daily, monthly, yearly, etc.) and applies aggregation functions (like mean()).

Visualization with Pandas

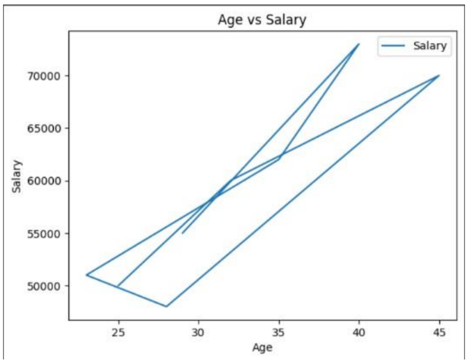
Basic Plotting using `df.plot()`

Pandas allows us to create simple plots using Matplotlib under the hood. We can visualize the relationship between different columns. For example, let's plot the **Age** and **Salary** columns.

```
import pandas as pd
import matplotlib.pyplot as plt

# Load the dataset
df_uploaded = pd.read_csv (r"/content/complete_employee_dataset (1).csv")

# Plot Age vs Salary
df_uploaded.plot(x='Age', y='Salary', kind='line')
plt.title('Age vs Salary')
plt.xlabel('Age')
plt.ylabel('Salary')
plt.show()
```



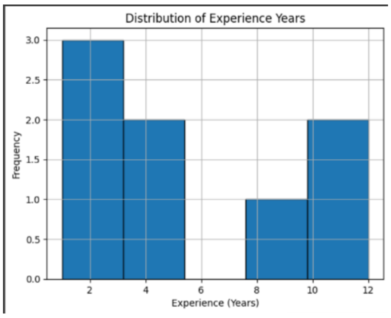
Histogram using df.hist()

A histogram allows us to visualize the distribution of numerical data. Here, we will plot the histogram of **Experience_Years** to understand its distribution across employees.

```
import pandas as pd
import matplotlib.pyplot as plt


df_uploaded = pd.read_csv(r"/content/complete_employee_dataset (1).csv")

# Plotting the histogram for Experience_Years
df_uploaded['Experience_Years'].hist(bins=5, edgecolor='black')
plt.title('Distribution of Experience Years')
plt.xlabel('Experience (Years)')
plt.ylabel('Frequency')
plt.show()
```



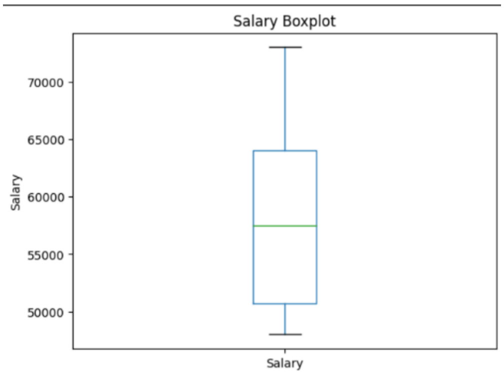
Boxplot using df.boxplot()

Boxplots are used to visualize the distribution of data and its outliers. We can plot a boxplot of **Salary** to see its spread and outliers

```
import pandas as pd
import matplotlib.pyplot as plt


df_uploaded = pd.read_csv(r"/content/complete_employee_dataset (1).csv")

df_uploaded.boxplot(column='Salary')
plt.title('Salary Boxplot')
plt.ylabel('Salary')
plt.grid(False)  # Optional: Remove the grid if not needed
plt.show()
```



Visualization Technique	Purpose	Output Type	Usage Example
Basic Plotting	To visualize data trends and relationships.	Line, bar, scatter plots, etc.	df.plot(x='column1', y='column2')
Histogram	To visualize the distribution of a dataset.	Bars representing frequency	df['column'].hist(bins=10)
Boxplot	To summarize the distribution and detect outliers.	Box with whiskers and points	df.boxplot(column='column')

Advanced Topics in Pandas

1. Window Functions

Window functions perform operations on a set of data points within a window defined over your dataset. These are particularly useful in time series data and allow for rolling or expanding computations on a DataFrame.

a) Rolling Window Operations

A rolling window operation calculates statistics over a rolling window of a fixed size. For example, you can calculate the rolling mean of a specific column.

Function:

```
df.rolling(window=n).mean()
```

window=n: Defines the size of the rolling window.

mean(): Can be replaced by other aggregation methods like .sum(), .min(), .max(), etc

Example:

```
# Rolling mean for Salary column with a window size of 3
df_uploaded['Rolling_Salary'] = df_uploaded['Salary'].rolling(window=3).mean()
print(df_uploaded[['Salary', 'Rolling_Salary']])
```

```
   Salary  Rolling_Salary
0   50000             NaN
1   60000             NaN
2   70000    60000.000000
3   48000    59333.333333
4   51000    56333.333333
5   62000    53666.666667
6   73000    62000.000000
7   55000    63333.333333
```

Expanding Window Operations

An expanding window starts with a single observation and gradually includes more as the window size increases. This operation accumulates values over time.

Function:

```
df.expanding(min_periods=1).sum()
```

Example:

```
# Expanding sum for Salary
df_uploaded['Expanding_Salary'] = df_uploaded['Salary'].expanding(min_periods=1).sum()
print(df_uploaded[['Salary', 'Expanding_Salary']])
```

```
   Salary  Expanding_Salary
0   50000           50000.0
1   60000          110000.0
2   70000          180000.0
3   48000          228000.0
4   51000          279000.0
5   62000          341000.0
6   73000          414000.0
7   55000          469000.0
```

Categorical Data

Converting columns to categorical data types is useful when working with non-numerical data like categories, which can save memory and improve performance.

Function:

```
df['category_column'] = df['category_column'].astype('category')
```

Example:

```
# Convert the 'Department' column to a categorical data type
df_uploaded['Department'] = df_uploaded['Department'].astype('category')
print(df_uploaded['Department'].dtype)
```

```
category
```

MultiIndex

MultiIndex (also called hierarchical index) allows you to index your DataFrame with multiple levels, which is especially useful for handling multi-dimensional data.

```
# Set a MultiIndex using the 'Age' and 'Experience_Years' columns
df_uploaded_multi = df_uploaded.set_index(['Age', 'Experience_Years'])
print(df_uploaded_multi.head())
```

```
                       Name  Salary   Department   Hire_Date  Bonus  \
Age Experience_Years
25  3                  John   50000           HR  2020-01-15   5000
32  8                  Jane   60000      Finance  2018-07-23   6000
45  10                  Doe   70000  Engineering  2016-03-12   7000
28  4                 Alice   48000    Marketing  2019-11-01   4800
23  2                   Bob   51000        Sales  2021-06-15   5100

                     Promotion_Eligibility  Projects_Completed Working_Remote  \
Age Experience_Years
25  3                                  Yes                   5             No
32  8                                   No                  12            Yes
45  10                                 Yes                  15             No
28  4                                  Yes                   8            Yes
23  2                                   No                   3             No

                      Rolling_Salary  Expanding_Salary
Age Experience_Years
25  3                            NaN           50000.0
32  8                            NaN          110000.0
45  10                  60000.000000          180000.0
28  4                   59333.333333          228000.0
23  2                   56333.333333          279000.0
```

Applying Functions in Pandas

1. df.apply():

The apply() function is used to apply a function along an axis (rows or columns) of a DataFrame. It allows element-wise, row-wise, or column-wise transformations.

- **Syntax:**

```
df.apply(function, axis=0)  # axis=0 applies function to each column, axis=1 applies t
```

```
def double_salary(x):
    return x * 2

df_uploaded['Salary_Double'] = df_uploaded['Salary'].apply(double_salary)
print(df_uploaded[['Salary', 'Salary_Double']].head())

   Salary  Salary_Double
0   50000         100000
1   60000         120000
2   70000         140000
3   48000          96000
4   51000         102000
```

df.applymap():

The applymap() function is used to apply a function to every element of the DataFrame, unlike apply(), which can work on rows or columns. It's particularly useful when you want to transform every single element in a DataFrame.

- **Syntax**:

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

```
def add_hundred(x):
    return x + 100

# Apply to all numeric data
df_uploaded_numeric = df_uploaded[['Salary', 'Experience_Years']].applymap(add_hundrec
print(df_uploaded_numeric.head())

   Salary  Experience_Years
0   50100               103
1   60100               108
2   70100               110
3   48100               104
4   51100               102
```

Lambda Functions:

Lambda functions are anonymous functions that are often used for simple transformations. You can use them inside apply() for quick, on-the-fly transformations.

- **Syntax**:

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

```
df_uploaded['Salary_Lambda'] = df_uploaded['Salary'].apply(lambda x: x * 2)
print(df_uploaded[['Salary', 'Salary_Lambda']].head())

   Salary  Salary_Lambda
0   50000         100000
1   60000         120000
2   70000         140000
3   48000          96000
4   51000         102000
```

Function	Scope	Best Use Case	Works With
apply()	Row-wise or column-wise	Apply a function to each row or column	DataFrame or Series
applymap()	Element-wise (entire DataFrame)	Apply a function to every element in the DataFrame	DataFrame
Lambda	Inline, concise function (with apply() or applymap())	Short, simple transformations within apply() or applymap()	DataFrame or Series