Import all Libraries - to executed everytime you open the notebook

In [ ]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import scipy.stats as stats
```

Installing the libraries if not present - one time activity

In [ ]:
```
!pip install numpy
!pip install pandas
!pip install matplotlib
!pip install seaborn
!pip install scipy
!pip install openpyxl
```

Dataframe

A DataFrame is two dimensional data structure where the data is arranged in the tabular format in rows and columns

DataFrame features:

- Columns can be of different data types
- Size of dataframe can be changes
- Axes(rows and columns) are labeled
- Arithmetic operations can be performed on rows and columns

Concataneting and Merging Dataframes

In [ ]:
```
df_jan = pd.DataFrame({"Order ID" : range(101, 111), "Sales" : np.random.randint(10
df_feb = pd.DataFrame({"Order ID" : range(111, 121), "Sales" : np.random.randint(10
df_mar = pd.DataFrame({"Order ID" : range(121, 131), "Sales" : np.random.randint(10
```

In [ ]:
```
df_jan.head(2)
```

In [ ]:
```
df_feb.head(2)
```

In [ ]:
```
df_mar.head(2)
```

Concatenate

pd.concat(`tuple of dfs` , `ignore_index = False` , `axis=0`)

In [ ]:
```
df = pd.concat((df_jan, df_feb, df_mar), ignore_index=True)
```

In [ ]:
```
df["Sales"].sum() # - total sales
```

In [ ]:
```
# Write the data to csv file -
df.to_csv("Sales.csv", index=None)
```

In [ ]:
```
# Write the data to excel file -
df.to_excel("Sales.xlsx", sheet_name="Total Sales", index=None)

# Note - when writing data to excel the original file must be closed.
```

Example

- Add a new column to each dataframe as month and value = "jan" or "feb" or "mar"
- Combine all the three dataframes and write to file

In [ ]:
```
df_jan["Month"] = "Jan"
df_feb["Month"] = "Feb"
df_mar["Month"] = "Mar"
```

In [ ]:
```
df = pd.concat((df_jan, df_feb, df_mar), ignore_index=True)
df.head(2)
```

In [ ]:
```
# Write the data to csv file - Always replace the exsisting data
df.to_csv("Sales.csv", index=None)
```

In [ ]:
```
# Write the data to csv file - Append data to exsisting file
df.to_csv("Sales.csv", index=None, mode = "a")
```

Merging Dataframes

`df1.merge(df2, how="", on = "", left_on="", right_on="")`

- **how** - type of merge (inner, left, right, outer)
- **on** - name of common column, used when both dfs have same name for the common/reference column
- **left_on** or **right_on** - name of left/right column when reference column names are different

In [ ]:
```
df_emp = pd.DataFrame({"Name" : ["Jack", "Bill", "Lizie", "Jane", "George"],
            "Designation" : ["HR", "Manager", "Developer", "Intern", "Manager"]})
df_emp
```

In [ ]:
```
base_salaries = pd.DataFrame({"Designation" : ["HR", "Developer", "Manager", "Senio
            "Salary": [40000, 25000, 70000, 1000000]})
base_salaries
```

Inner Merge

- Gives data only for the common values for reference column in both the dfs

In [ ]:
```
df_emp.merge(base_salaries, on="Designation", how = "inner" )
```

Left Merge

- Gives data for the left table and corresponding values from right table based on reference column. Gives null for missing values

In [ ]:
```
df_emp.merge(base_salaries, on="Designation", how = "left" )
```

Right Merge

- Gives data for the right table and corresponding values from left table based on reference column. Gives null for missing values

In [ ]:
```
df_emp.merge(base_salaries, on="Designation", how = "right" )
```

Outer Merge

In [ ]:
```
df_emp.merge(base_salaries, on="Designation", how = "outer" )
```

Examples

Ex. Calculate total sales across all three months using Excel plug-in

In [ ]:
```
```

Ex. Create a table displaying salary of each employee

In [ ]:
```
```

DataFrame toolkit -

Ex. Read data from `BSE Sensex 30 Historical Data.csv`

In [ ]:
```
df = pd.read_csv(r"./Datasets/BSE Sensex 30 Historical Data.csv")
df.head(2)
```

Drop a column or row from dataframe

`df.drop(columns = [], index = [], inplace=False)`

- inplace = False returns a new DataFrame (default), True modifies original df

In [ ]:
```
df.drop(columns=["High", "Low"], index=[0, 10, 4, 8])
```

Working with **null** values

`df.isna()` - Detect missing values. Return a boolean same-sized object indicating if the values are NA.

`df.fillna(value=None, inplace=False, method = None)` - Fill NA/NaN values using the specified method.

method : {'backfill', 'bfill', 'ffill', None}

```
In [ ]: df.isna().any()  # True means there is atleast 1 null value in the column
```

Incase entire row/column is null - Drop null rows

df.dropna(`axis = 0` , `how = "any"` , `inplace = False`)

- axis 0 for row or 1 for column
- how - {any or all}

```
In [ ]: df.shape
```

```
In [ ]: # df.dropna(axis = 0, how = "any") - deletes rows with any 1 null value
        df.dropna()
```

```
In [ ]: df.dropna(axis= 0, how="all", inplace=True) # - deletes rows with with all null val
```

```
In [ ]: df.isna().any() # null rows are deleted by vol column still has null values
```

```
In [ ]: df.dropna(axis= 1, how="any") # - deletes column with any 1 null values
```

```
In [ ]: df.dropna(axis= 1, how="all", inplace=True) # - deletes column with any 1 null valu
```

```
In [ ]: df.head(2)
```

Extracting null rows for Vol column

```
In [ ]: df[df.isna().any(axis = 1)] # for any column
```

```
In [ ]: df[df["Vol."].isna()] # for specific column
```

Ex. Replace the null value with default

```
In [ ]: df["Vol."].fillna(0, inplace=True)  # syntax in older pandas version
```

```
In [ ]: df.fillna({"Vol.": 0, "High" : 1, "Low" : df.Low.mean()})  # new syntax - provides
```

Ex. Replace null with ffill or bfill

```
In [ ]: df["Vol."] = df["Vol."].ffill()  # fowardfill avoid inplace = True in this case
```

```
In [ ]: df["Vol."] = df["Vol."].bfill()  # backwardfill avoid inplace = True in this case
```

Removing Duplicate Data

In [86]:
```
df.duplicated().any()
```

Out[86]:
```
np.False_
```

`df.drop_duplicates(subset = [columns], inplace=False)`

In [88]:
```
df.drop_duplicates(inplace=True)
```

Replacing values

df.replace({ `colname` : { `old_value` : `new_value` }}, `inplace=True`)

In [89]:
```
df.head()
```

Out[89]:

	Date	Price	Open	High	Low	Vol.	Change %
0	16-04-2025	76,761.72	76,996.78	76,996.78	76,544.07	4.99M	0.03%
1	15-04-2025	76,734.89	76,852.06	76,857.05	76,449.56	12.93M	2.10%
2	11-04-2025	75,157.26	74,835.49	75,467.33	74,762.84	14.23M	1.77%
3	09-04-2025	73,847.15	74,103.83	74,103.83	73,673.06	9.15M	-0.51%
4	08-04-2025	74,227.08	74,013.73	74,859.39	73,424.92	17.06M	1.49%

In [ ]:
```
df.replace({"Change %" : {"0.03%" : "0.05%"}})
```

Clean the dataset

In [91]:
```
df.dtypes
```

Out[91]:
```
Date         object
Price        object
Open         object
High         object
Low          object
Vol.         object
Change %     object
dtype: object
```

In [130…
```
df["Price"] = df["Price"].str.replace(",", "").astype(float)
df["Open"] = df["Open"].str.replace(",", "").astype(float)
df["High"] = df["High"].str.replace(",", "").astype(float)
df["Low"] = df["Low"].str.replace(",", "").astype(float)
df["Change %"] = df["Change %"].str.replace("%", "").astype(float)
```

In [131…
```
df["Volume"] = df["Vol."].str[:-1].astype(float)
df["temp"] = df["Vol."].str[-1]
```

```
df["Volume"] = df["temp"].map({"M" : 1000000, "K" : 1000, "B" : 1000000000}) * df["
df.head(10)
```

Out[131…

	Date	Price	Open	High	Low	Vol.	Change %	Volume	temp
0	2025-04-16	76761.72	76996.78	76996.78	76544.07	4.99M	0.03	4990000.0	M
1	2025-04-15	76734.89	76852.06	76857.05	76449.56	12.93M	2.10	12930000.0	M
2	2025-11-04	75157.26	74835.49	75467.33	74762.84	14.23M	1.77	14230000.0	M
3	2025-09-04	73847.15	74103.83	74103.83	73673.06	9.15M	-0.51	9150000.0	M
4	2025-08-04	74227.08	74013.73	74859.39	73424.92	17.06M	1.49	17060000.0	M
5	2025-07-04	73137.90	71449.94	73403.99	71425.01	29.37M	-2.95	29370000.0	M
6	2025-04-04	75364.69	76160.09	76258.12	75240.55	29.37M	-1.22	29370000.0	M
7	2025-03-04	76295.36	75811.86	76493.74	75807.55	6.92M	-0.42	6920000.0	M
8	2025-02-04	76617.44	76146.28	76680.35	76064.94	10.75M	0.78	10750000.0	M
9	2025-01-04	76024.51	76882.58	77487.05	75912.18	10.59M	-1.80	10590000.0	M

Grouping Dataframes

`df.groupby(by=None, as_index=True, sort=True, dropna=True)`

- use of `agg()`

In [133…

```
df["Date"] = pd.to_datetime(df["Date"], format = "mixed")
df.insert(1, "Year", df["Date"].dt.year)
df.insert(2, "Month", df["Date"].dt.month_name())
df.insert(3, "Month#", df["Date"].dt.month)
df.head()
```

Out[133…

	Date	Year	Month	Month#	Price	Open	High	Low	Vol.	Chan
0	2025-04-16	2025	April	4	76761.72	76996.78	76996.78	76544.07	4.99M	0.
1	2025-04-15	2025	April	4	76734.89	76852.06	76857.05	76449.56	12.93M	2.
2	2025-11-04	2025	November	11	75157.26	74835.49	75467.33	74762.84	14.23M	1.
3	2025-09-04	2025	September	9	73847.15	74103.83	74103.83	73673.06	9.15M	-0.
4	2025-08-04	2025	August	8	74227.08	74013.73	74859.39	73424.92	17.06M	1.

In [134…

```
df.Year.unique()
```

Out[134…

```
array([2025, 2024, 2023], dtype=int32)
```

Ex. Year average Price

In [136…

```
df.groupby("Year")["Price"].mean().round(2)
```

Out[136…

```
Year
2023    64567.91
2024    77225.52
2025    76175.51
Name: Price, dtype: float64
```

In [138…

```
df.groupby(["Year", "Month#", "Month"])["Price"].mean().round(2)
```

```
Out[138…  Year  Month#  Month
          2023  1       January      65069.51
                2       February     63441.38
                3       March        59872.70
                4       April        61990.71
                5       May          62860.83
                6       June         63941.57
                7       July         66308.81
                8       August       65205.26
                9       September    65859.63
                10      October      64618.01
                11      November     65714.73
                12      December     68690.92
          2024  1       January      73811.38
                2       February     74216.63
                3       March        74811.65
                4       April        75312.02
                5       May          75225.31
                6       June         77504.34
                7       July         79309.00
                8       August       79210.81
                9       September    81191.39
                10      October      79498.66
                11      November     78482.14
                12      December     78356.19
          2025  1       January      76630.13
                2       February     75727.45
                3       March        76407.02
                4       April        76087.01
                5       May          76000.76
                6       June         76787.75
                7       July         75882.45
                8       August       76187.78
                9       September    75733.68
                10      October      76268.63
                11      November     75184.39
                12      December     75100.42
          Name: Price, dtype: float64
```

Ranking and Sorting Dataframes

Ex. Rank the products in descending order of `Sales`

```
In [ ]:
```

Ex. Sort the data in ascending order of `Rank`

```
In [ ]:
```

Setting and Resetting Index

`df.set_index(keys, drop=True, inplace=False,)` - Set the DataFrame index (row labels) using one or more existing columns or arrays (of the correct length). The index can replace the existing index or expand on it.

In [ ]:

`df.reset_index(level=None, drop=False, inplace=False,)` - Reset the index of the DataFrame, and use the default one instead. If the DataFrame has a MultiIndex, this method can remove one or more levels.

In [ ]:

Working with dates

Create columns Year and Month - extract data using pd.DatetimeIndex

In [ ]:

Extract data for 2023

In [ ]:

In [ ]:

In [ ]:

In [ ]:

Ex. Visualise Trend and Sesonality of the data

In [ ]:

Extract data for Jan - 2023

In [ ]:

Extract data for Jan - 2023 and 2024

In [ ]:

Extract data starting from April - 2024

In [ ]:

Extract data from Jan-2023 to Apr-2024

In [ ]:

Descriptive Statistics

Descriptive statistics deals with summarizing and describing the features of a dataset or sample. Descriptive statistics provides a summary of the main features of the data, including measures of central tendency, dispersion, shape, and relationships between variables.

Measures of Central Tendency:

```
- Mean: The average value of the data points.
- Median: The middle value of the data when arranged in ascending
order.
- Mode: The most frequently occurring value in the dataset.
```

Measures of Dispersion:

```
- Range: The difference between the maximum and minimum values in
the dataset.
- Variance: The average of the squared differences from the mean.
- Standard Deviation: The square root of the variance, representing
the average deviation from the mean.
```

Measures of Shape:

```
- Skewness: A measure of the asymmetry of the distribution.
    - Positive skewness indicates a longer right tail and a
concentration of data on the left side.
    - Negative skewness indicates a longer left tail and a
concentration of data on the right side.
    - Skewness close to zero indicates approximate symmetry around
the mean.

- Kurtosis: A measure of the "peakedness" or "flatness" of the
distribution.
    - Positive kurtosis indicates heavy tails and a sharp peak
(leptokurtic).
    - Negative kurtosis indicates light tails and a flat peak
(platykurtic).
    - A kurtosis of 0 indicates a distribution with similar tails
to the normal distribution (mesokurtic).
```

Frequency Distribution:

```
- Frequency table: A table that shows the frequency or count of
each value in the dataset.
- Histogram: A graphical representation of the frequency
distribution, showing the distribution of values in bins or
intervals.
```

Measures of Association:

```
- Correlation: A measure of the strength and direction of the
linear relationship between two variables.
```

```
- Covariance: A measure of the joint variability between two
variables.
```

In [ ]:
```python
# dataset consists of weights children in the age group of 0 to 10 years
weights = np.array([20.8,15.3,23.2,15.5,17.5,27.3,23.3,20.5,16.4,17.4,22.6,20.8,16.
```

In [ ]:

In [ ]:

In [ ]:

In [ ]:
```python
# dataset consists of Salaries of employees in an organisation
salaries = np.array([29756,20014,20347,57214,41327,40209,93390,122004,17725,47210,4
```

In [ ]:

In [ ]:

Handling Outliers -

`Z-Score Method:`

- The z-score method involves calculating the z-score for each data point, which represents the number of standard deviations away from the mean. Data points with z-scores beyond a certain threshold (e.g., |z-score| > 3) are considered outliers and can be removed or treated separately. The z-score method is sensitive to the mean and standard deviation of the data, and it assumes that the data is normally distributed. This method is useful when the data is approximately normally distributed and when the goal is to identify outliers based on their deviation from the mean.

`IQR Method:`

- The IQR method involves calculating the interquartile range (IQR), which is the difference between the third quartile (Q3) and the first quartile (Q1) of the data. Outliers are defined as data points that fall below Q1 - 1.5 * IQR or above Q3 + 1.5 * IQR. The IQR method is robust to outliers and does not assume any specific distribution of the data. This method is useful when the data is skewed or not normally distributed, as it focuses on the middle 50% of the data and is less influenced by extreme values. In general, if the data is approximately normally distributed and the goal is to identify outliers based on their deviation from the mean, the z-score method may be more appropriate. On the other hand, if the data is skewed or not normally distributed, or if the goal is to identify outliers based on their relative position within the dataset, the IQR method may be a better choice.

In [ ]:

In [ ]:

```
```

In [ ]:

```
```

In [ ]:

```
```