## Importing, Reading, and Creating DataFrames:

1. Importing Pandas : ***import pandas as pd***

2. Reading Data from Different Formats:
   - CSV: ***df = pd.read_csv('file.csv')***
   - Excel: ***df = pd.read_excel('file.xlsx')***
   - JSON: df = ***pd.read_json('file.json')***
   - SQL: ***df = pd.read_sql(query, connection_object)***

3. Creating a DataFrame:
   - From a dictionary:

```python
data = {'col1': [1, 2], 'col2': [3, 4]}
df = pd.DataFrame(data)
```

   - From a list of lists:

```python
data = [[1, 2], [3, 4]]
df = pd.DataFrame(data, columns=['col1', 'col2'])
```

   - From a NumPy array:

```python
import numpy as np
data = np.array([[1, 2], [3, 4]])
df = pd.DataFrame(data, columns=['col1', 'col2'])
```

4. Inspecting DataFrame:
   - Display top rows:

```python
df.head()
```

   - Display bottom rows:

```python
df.tail()
```

   - Show DataFrame info:

```python
df.info()
```

   - Describe summary statistics:

```python
df.describe()
```

5. Filtering DataFrame:
   - Filter rows based on column values:

```python
df[df['col1'] > 1]
```

   - Filter rows based on string content:

```python
df[df['Country'].str.contains('United')]
```

6. Handling Missing Data:
- Drop missing values:
  ***df.dropna()***

- Fill missing values:
  df.fillna(value)

7. Renaming Columns:
***df.rename(columns={'old_name': 'new_name'}, inplace=True)***

8. Adding/Removing Columns:
- Add a new column:
  ***df['new_col'] = value***

- Remove a column:
  ***df.drop('col1', axis=1, inplace=True)***

9. Sorting Data:
- Sort by a column:
  ***df.sort_values('col1', ascending=False)***

10. Grouping Data:
- Group by a column and apply aggregation:
  ***df.groupby('col1').sum()***

11. Merging DataFrames:
- Merge two DataFrames on a common column:
  ***pd.merge(df1, df2, on='common_col')***

- Concatenate DataFrames along rows or columns:
  ***pd.concat([df1, df2], axis=0)***

12. Exporting DataFrame to File:
- To CSV:
  ***df.to_csv('output.csv', index=False)***

- To Excel:
  ***df.to_excel('output.xlsx', index=False)***

- To JSON:
  ***df.to_json('output.json')***

## Viewing & Selecting DataFrames:

- View the first or last few rows of a DataFrame:
  ***df.head()*** # First 5 rows by default
  ***df.tail(3)*** # Last 3 rows

- View specific columns:
  ***df['col1']***
  ***df[['col1', 'col2']]***

- Select rows by index position:
  ***df.iloc[0]*** # First row
  ***df.iloc[0:5]*** # First 5 rows
  ***df.iloc[:, 0]*** # First column
  ***df.iloc[0:5, 0:3]*** # First 5 rows and first 3 columns

- Select rows by label:
  ***df.loc['row_label']***
  ***df.loc[:, 'col1']*** # All rows, specific column

- View summary statistics of a DataFrame:
  ***df.describe()***

- View the DataFrame's index and columns:
  ***df.index***
  ***df.columns***

- View data types of columns:
  ***df.dtypes***

- Transpose a DataFrame:
  ***df.T***

## Filtering, Sorting, and Indexing:

- Filter rows by column value:
  ***df[df['col1'] > 10]***

- Filter rows by multiple conditions:
  ***df[(df['col1'] > 10) & (df['col2'] < 5)]***

- Filter rows based on string content:
  ***df[df['col1'].str.contains('substring')]***

- Filter rows by missing values:
  ***df[df['col1'].isna()]***

- Sorting DataFrame by column:
  ***df.sort_values('col1')***

- Sorting DataFrame in descending order:
  ***df.sort_values('col1', ascending=False)***

- Sorting by multiple columns:
  ***df.sort_values(['col1', 'col2'], ascending=[True, False])***

- Setting a column as the index:
  ***df.set_index('col1', inplace=True)***

- Reset the index of a DataFrame:
  ***df.reset_index()***

- Selecting data by index:
  ***df.loc['index_label']*** # Select row by index label
  ***df.loc['start_label':'end_label']*** # Select rows in a range of index labels

- Multi-level indexing (hierarchical indexing):
  ***df.set_index(['col1', 'col2'], inplace=True)***

- Filter data with a multi-level index:
  ***df.xs('label', level='col1')*** # Select data at specific level of index

## Grouping, Aggregating, Transforming:

### 1. Grouping Data:

- Group by a single column:
  ***df.groupby('column_name')***

- Group by multiple columns:
  ***df.groupby(['col1', 'col2'])***

- Apply aggregation after grouping:
  ***df.groupby('col1').sum()***
  ***df.groupby('col1').mean()***
  ***df.groupby('col1').size()***

- Group by and apply multiple aggregations:
  ***df.groupby('col1').agg(['sum', 'mean'])***

- Access grouped data:
  ***grouped = df.groupby('col1')***
  ***grouped.get_group('value')***

- Grouping with custom aggregations:
  ***df.groupby('col1').agg({'col2': 'sum', 'col3': 'mean'})***

### 2. Aggregating and Applying Functions:

- Aggregation methods:
  ***df['column_name'].sum()***
  ***df['column_name'].mean()***
  ***df['column_name'].count()***

- Apply a function across columns:
  ***df.apply(np.sqrt)***

- Apply custom functions to DataFrame:

```
def custom_func(x):
    return x + 1

df['new_column'] = df['existing_column'].apply(custom_func)
```

- Transforming data:
  ***df.groupby('col1')['col2'].transform(lambda x: x - x.mean())***

- Applying element-wise functions:
  ***df['new_col'] = df['existing_col'].map(lambda x: x*2)***

- Using lambda expressions with apply:
  ***df['col1'] = df['col1'].apply(lambda x: x*2 if x > 5 else x)***

- Apply different functions to different columns:
  ***df.agg({'col1': 'mean', 'col2': 'sum'})***

### 3. Transforming:

The transform() method allows you to execute a function for each value of the DataFrame.

***dataframe.transform(func, axis, raw, result_type, args, kwds)***

```
df.groupby('order')["ext price"].transform('sum')
```

```
df["Order_Total"] = df.groupby('order')["ext price"].transform('sum')

df["Percent_of_Order"] = df["ext price"] / df["Order_Total"]
```

### 4. Merging & Concatenating DataFrames:

- `merge()` **for combining data on common columns or indices**
- `.join()` **for combining data on a key column or an index**
- `concat()` **for combining DataFrames across rows or columns**

- Merge two DataFrames on a common column:
***pd.merge(df1, df2, on='common_column')***
- Merge with multiple keys:
***pd.merge(df1, df2, on=['key1', 'key2'])***

- Merge with inner, outer, left, and right joins:
***pd.merge(df1, df2, on='common_column', how='inner') # 'left', 'right', 'outer'***

- Merge with suffixes for overlapping columns:
***pd.merge(df1, df2, on='common_column', suffixes=('_left', '_right'))***

- Concatenate DataFrames along rows (vertically):
***pd.concat([df1, df2], axis=0)***

- Concatenate DataFrames along columns (horizontally):
***pd.concat([df1, df2], axis=1)***

- Concatenating with keys (for multi-level indexing):
***pd.concat([df1, df2], keys=['key1', 'key2'])***

- Join DataFrames on indexes:
***df1.join(df2, how='inner') # Can also be 'outer', 'left', 'right'***

# Handling missing data :

## Import Pandas:

- `import pandas as pd` imports the Pandas library and assigns the alias `pd` for convenience.

## Read CSV Data:

- `df=pd.read_csv("C:/Users/PANDAS_DATA/pandastutorial-main/pandastutorial-main/Datasets/sample.csv")` reads the CSV file into a DataFrame named `df`, assuming the specified path exists.

## Previewing Data:

- `print(df.head())` displays the first few rows of `df` to get a glimpse of the data.

## Identifying Missing Values:

- `print(df.isnull())` creates a DataFrame showing Boolean values (True/False) indicating where missing values (NaN) exist.

## Counting Missing Values per Column:

- `print(df.isnull().sum())` calculates the number of missing values in each column and displays the results.

## Total Missing Values:

- `total_missing = df.isnull().sum().sum()` computes the total number of missing values across all columns in `df`.

## Original DataFrame Shape:

- `print(df.shape)` displays the original dimensions (number of rows, columns) of `df`.

## Dropping Rows with Any Missing Values:

- `df2 = df.dropna()` creates a new DataFrame `df2` by removing rows containing any missing values (default behaviour).
- `print(df2.shape)` shows the potentially reduced number of rows in `df2`.

## Dropping Columns with Any Missing Values:

- `df3 = df.dropna(axis=1)` creates `df3` by dropping columns with at least one missing value.
- `print(df3.shape)` displays the potentially reduced number of columns in `df3`.

## Dropping Rows Only if All Values Are Missing:

- `df4 = df.dropna(how='all')` creates `df4` by dropping rows where all values are missing (NaN).
- `print(df4.shape)` shows the shape of `df4`, which might be the same as `df` if no rows have all-NaN values.

## Modifying Inplace:

- `df.dropna(inplace=True)` directly modifies `df` to remove rows with missing values. This avoids creating copies.
- `print(df.shape)` displays the updated shape of `df`, which will be the same as `df2`.

## Key Points:

- `dropna()` offers flexibility in handling missing values by specifying `axis` (rows or columns) and `how` (any or all missing values per row).
- Consider the implications of data loss when dropping rows