1 Data Structures :

- **Series**: 1D labeled array, like a column
  - s = pd.Series([1, 2, 3])
- **DataFrame**: 2D labeled data structure, like a table with rows and columns.
  - df = pd.DataFrame({'A': [1, 2], 'B': [3, 4]})

Reading & Writing Data :

- df = pd.read_csv('file.csv')   # Read CSV
- df.to_csv('output.csv', index=False) # Write CSV
- df = pd.read_excel('file.xlsx')   # Excel
- df.to_excel('output.xlsx', index=False)

Viewing and Inspecting Data :

- df.head()   # First 5 rows
- df.tail()   # Last 5 rows
- df.info()   # Info on columns, non-null count, dtypes
- df.describe()   # Summary stats for numeric columns
- df.shape   # Rows and columns
- df.columns   # List column names
- df.dtypes   # Data types of columns

Data Selection :

- [ ] [ ] ; loc ; iloc
- Select * from data
  - movies[:][:]
  - movies.loc[:,:]
  - Movies.iloc[:,:]
- select first 5 rows from the data using all methods
  - loc - Label (row/column names),Yes for slicing (end included)
  - iloc - Integer position (row/column index), No for slicing (end excluded)
  - movies[0:5][:]
  - movies.loc[0:4,:]
  - movies.iloc[0:5,:]
- select first 5 rows, 2 columns in the dataset
  - movies.iloc[0:5,0:2]
- select first 10 rows, and type, title , director columns
  - Movies[0:10][["type","title","director"]]
  - movies.loc[0:9, ["type","title","director"]]

Filtering Data :

- select * where type = movie and country = India
  - movies.loc[(movies["type"]=="Movie") & (movies["country"] == "India"),:]
- Select distinct country
  - movies.loc[:,"country"].unique()

- Select distinct country,type combinations
    - movies.loc[:,["country","type"]].drop_duplicates()

Handling Missing Values :

- Get the number of missing values in each column
    - movies.isnull().sum()
        - movies.isnull() gives true false values in all the cells. It has the same shape as of the dataframe
        - movies.isnull().sum() counts the true values
- Drop all rows containing missing values
    - movies.dropna()
- Replace all missing (NaN) values in the DataFrame df with 0
    - movies.fillna(0)

Iterating :

- iterrows() is a Pandas DataFrame method that lets you iterate over the rows of a DataFrame one by one.
- It returns each row as a (index, Series) pair.
    - Below prints each row one by one. Each row is a pandas series

    -
      ```
      for index, row in movies.iterrows():
         print(index)
         print(row)
         print(type(row))
      ```
- dict.items() - Returns a view of **(key, value)** pairs in the dictionary.

    -
      ```
      student = {"name": "Alice", "age": 25}
      for key, value in student.items():
         print(key, "→", value)
      output
      name → Alice
      age → 25
      ```

Aggregations :

- select country, count(showid) from movies where type = movie

    -
      ```
      df =movies.loc[movies["type"]=="Movie",:]
      df.groupby("country")["show_id"].count()
      ```
- select country, release year count(showid) as show_count, count(director) as director_count, countd(director) as director_distinct_count from movies where type = movie

    -
      ```
      result = df.groupby(["country","release_year"]).agg({"show_id" : ["count"],
      "director" : ["count","nunique"]})
      result.columns = ["show_count","director_count","director_distinct_count"]
      result.reset_index()
      ```

Merging, and Joins :

- select * from movies where country = india union select * from movies where country = 'United States'

    -
      ```
      df1 = movies.loc[movies["country"]== "India",:]
      df2 = movies.loc[movies["country"]== "United States",:]
      df3 = pd.concat([df1,df2])
      df3
      ```
- create two dataframes, and join them (inner, left, right, outer)

    -
      ```
      movies_sample = pd.DataFrame({
         'movie_id': [1, 2, 3, 4],
         'title': ['Inception', 'Titanic', 'Avatar', 'Interstellar'],
         'director_id': [101, 102, 102, 103]
      })
      directors = pd.DataFrame({
         'director_id': [101, 102, 104],
         'name': ['Christopher Nolan', 'James Cameron', 'Steven Spielberg']
      })
      inner = pd.merge(movies_sample, directors, on='director_id', how='inner')
      inner
      ```

Sorting :

- select year, country, type, show_id from movies order by year desc, country asc

    -
      ```
      movies.loc[:,["release_year","country","type","show_id"]].sort_values(by = ["release_year","country"], ascending = [False,True])
      ```
- Rank, Dense Rank, Row Number

Apply and Lambda

- The .apply() method applies a function to each group (if used with groupby), or to each row/column (if used on a DataFrame), or element (if used on a Series).
- Understanding what is getting passed to apply is important.
- df.apply(lambda x : some logic) --->> logic gets applied on each column of df
    - Converts varchar columns to uppercase

    -
      ```
      df = pd.DataFrame({
         'name': ['Alice', 'Bob', 'Charlie'],
         'id' : ['a','b','c'],
         'math': [80, 90, 70],
         'science': [85, 88, 95]
      })

      df.select_dtypes(include='object').apply(lambda x: x.str.upper())
      ```

    - Same code can be written this way as well to see what is happening, and what is passed to x

        -
          ```
          def uppercase(x):
          ```

```
          print(x)
          print(type(x))
          print(x.str.upper())
          print("end")

        df.select_dtypes(include='object').apply(uppercase)
```

- Both .map() and .apply() can apply a function to each value in a Series. But map() is slightly faster with series
  - .apply() does not accept a dictionary when used on a pandas Series. Only .map() supports direct dictionary mapping.
  - titanic["sex"].map({'male': 0, 'female': 1})
  - .map() is designed to map values directly using a dictionary, Series, or function.
  - .apply() expects a function, not a dictionary.
- df["name"].apply(lambda x: x.upper())
  - Here each value of the name column gets passed one by one
  - We can write a similar function like above to see that.
- Assign 1 if a , 2 if b, else 0 on id column

  -
    ```
    def assign_flag(x):
      print(x)
      print(type(x))
      if x == 'a':
        print("end")
        return 1
      elif x == 'b':
        print("end")
        return 2
      else:
        print("end")
        return 0
    df["id"].apply(assign_flag)
    ```

- df.apply(lambda x : some logic, axis = 1) ---->> logic gets applied on each row of df. Here each row gets passed one by one.
  - Try one example
  - We can write functions similar to above
- Apply with groupby

  -
    ```
    df = pd.DataFrame({
        'Department': ['HR', 'HR', 'IT', 'IT'],
        'Employee': ['Alice', 'Bob', 'Charlie', 'David'],
        'Salary': [50000, 55000, 70000, 72000]
     })
    ```
  - df.groupby('Department')['Salary'].apply(lambda x: x.mean())

  -
    ```
    def group_dep(x) :
        print(x)
    ```

```
    print(type(x))
    print("end")
    return x.mean()
mean = df.groupby('Department')['Salary'].apply(group_dep)
print(mean)
```

  - groupby('Department') splits the DataFrame into two groups: HR and IT.
  - .apply(lambda x: x.mean()) computes the mean salary for each group.

- .applymap() is used to apply a function elementwise to every cell in a DataFrame only.
  - It does not work with Series or GroupBy objects.