Practical No. 1

Aim:
Load a CSV dataset into a Pandas DataFrame. Perform basic data inspection such as displaying the first few rows, checking data types, handling missing values, and removing duplicate rows.

Theory:
Data inspection is the first step in data analysis where raw data is checked for correctness.
Pandas allows loading external datasets like CSV into DataFrames for easy handling.
Functions like `head()`, `dtypes`, and `isnull()` help to understand structure, types, and missing values.
Cleaning operations like removing duplicates improve data quality for further processing.

Input (Example Code):

```
import os
import pandas as pd

fname = 'data.csv'  # change this to the correct path if needed

if not os.path.exists(fname):
    print(f"File '{fname}' not found. Creating a demo DataFrame instead.")
    df = pd.DataFrame({
        'name': ['divya', 'prachi', 'yash', 'yogesh'],
        'score': [85, 92, 78, 92]
    })
else:
    df = pd.read_csv(fname)

print("HEAD:")
print(df.head(), "\n")

print("DTYPES:")
print(df.dtypes, "\n")

print("MISSING VALUES (per column):")
print(df.isnull().sum(), "\n")

df = df.drop_duplicates()
print("After drop_duplicates():")
print(df)
```

Output :

```
File 'data.csv' not found. Creating a demo DataFrame instead.
HEAD:
     name  score
0   divya     85
1  prachi     92
2    yash     78
3  yogesh     92

DTYPES:
name     object
score     int64
dtype: object

MISSING VALUES (per column):
name     0
score    0
dtype: int64

After drop_duplicates():
     name  score
0   divya     85
1  prachi     92
2    yash     78
3  yogesh     92
```

Practical No. 2

AIM:

To load a dataset using Pandas and perform column-wise operations such as adding a calculated column, renaming columns, changing data types, and dropping unnecessary columns.

Theory:

Pandas supports creation of new calculated columns using arithmetic expressions.
It provides renaming and datatype conversion functions for consistent formatting.
Unnecessary columns can be dropped to simplify the dataset.
Such operations are important for feature engineering in data analytics.

Input (Example Code):

```
import pandas as pd

# Sample dataset (You can replace it using read_csv('data.csv'))
df = pd.DataFrame({
    'name': ['Alice', 'Bob', 'Carol'],
    'math': [80, 90, 85],
    'science': [88, 92, 79]
})

print("Original DataFrame:")
print(df, "\n")

# 1️⃣ Add a new calculated column - Total Marks
df['total'] = df['math'] + df['science']
print("After adding 'total' column:")
print(df, "\n")

# 2️⃣ Rename columns
df = df.rename(columns={'math': 'Mathematics', 'science': 'Science'})
print("After renaming columns:")
print(df, "\n")

# 3️⃣ Change data type of 'total' column to float
df['total'] = df['total'].astype(float)
print("After changing data type of 'total':")
print(df.dtypes, "\n")

# 4️⃣ Drop an unnecessary column
df = df.drop(columns=['Science'])
print("After dropping 'Science' column:")
print(df)
```

Output:

```
Original DataFrame:
    name  math  science
0  Alice    80       88
1    Bob    90       92
2  Carol    85       79

After adding 'total' column:
    name  math  science  total
0  Alice    80       88    168
1    Bob    90       92    182
2  Carol    85       79    164

After renaming columns:
    name  Mathematics  Science  total
0  Alice           80       88    168
1    Bob           90       92    182
2  Carol           85       79    164

After changing data type of 'total':
name            object
Mathematics      int64
Science          int64
total          float64
dtype: object

After dropping 'Science' column:
    name  Mathematics  total
0  Alice           80  168.0
1    Bob           90  182.0
2  Carol           85  164.0
```

Practical No. 3

AIM:

Using NumPy, generate a random dataset of 1000 values. Calculate basic statistical measures such as mean, median, variance, standard deviation, minimum, and maximum using NumPy functions.

Theory:

NumPy is a powerful numerical computing library used for mathematical and statistical analysis.
Random arrays allow sample data generation for experiments and simulations.
Statistical measures like mean, median, variance, and standard deviation help understand data distribution.
NumPy performs computation in a fast and memory-efficient manner.

Input (Example Code):

```
import numpy as np

# Generate a random dataset of 1000 values
data = np.random.randn(1000)  # Normally distributed data

print("Dataset (first 10 values):")
print(data[:10], "\n")  # Show sample

# Statistical calculations
mean_val = np.mean(data)
median_val = np.median(data)
variance_val = np.var(data)
std_val = np.std(data)
min_val = np.min(data)
max_val = np.max(data)

print("Mean:", mean_val)
print("Median:", median_val)
print("Variance:", variance_val)
print("Standard Deviation:", std_val)
print("Minimum:", min_val)
print("Maximum:", max_val)
```

Output:

```
Dataset (first 10 values):
[-1.66599931  0.05784512  1.31966176 -0.0020768   0.83739217 -0.7500639
 -0.04565545 -0.15656713  0.30850158 -0.1973774 ]

Mean: 0.00390461371311154 46
Median: 0.0146821947624599 66
Variance: 1.0449336144252248
Standard Deviation: 1.022219944251346
Minimum: -3.4029368234663826
Maximum: 3.42002277436092
```

Practical No. 4

AIM:

Write a Pandas program to group a dataset by one or more categorical columns and calculate summary statistics such as count, mean, and standard deviation for each group.

Theory:

Grouping allows data to be divided into categories for better summarization.
The `groupby()` function calculates statistics such as count, mean, and standard deviation for each group.
It helps in comparing different segments of data like departments, products, or regions.
This is widely used in reporting, business analytics, and decision-making.

Input (Example Code):

```
import pandas as pd

# Sample dataset with categorical column 'Department'
df = pd.DataFrame({
    'Department': ['IT', 'IT', 'HR', 'HR', 'Sales', 'Sales'],
    'Employee': ['A', 'B', 'C', 'D', 'E', 'F'],
    'Salary': [45000, 50000, 42000, 43000, 47000, 48000]
})

print("Original DataFrame:")
print(df, "\n")

# Grouping by Department and calculating summary statistics
group_stats = df.groupby('Department')['Salary'].agg(['count', 'mean',
'std'])

print("Grouped Summary Statistics:")
print(group_stats)
```

Output:

```
Original DataFrame:
  Department Employee  Salary
0         IT        A   45000
1         IT        B   50000
2         HR        C   42000
3         HR        D   43000
4      Sales        E   47000
5      Sales        F   48000

Grouped Summary Statistics:
            count     mean          std
Department
HR              2  42500.0   707.106781
IT              2  47500.0  3535.533906
Sales           2  47500.0   707.106781
```

Practical No. 5

AIM:

Load a dataset into Pandas and filter rows based on complex conditions using `.loc` and `.query()`.

Theory:

Filtering extracts only the required records from a dataset based on conditions.
`.loc()` helps filtering using logical indexing on DataFrame rows and columns.
`.query()` uses simple SQL-style expressions for readability and faster execution.
Filtering helps focus on meaningful information and remove irrelevant data.

Input (Example Code):

```
import pandas as pd

# Sample dataset
df = pd.DataFrame({
    'Name': ['Alice', 'Bob', 'Carol', 'David', 'Eva'],
    'Age': [24, 30, 22, 28, 35],
    'Score': [88, 75, 92, 60, 85],
    'Department': ['IT', 'HR', 'IT', 'Sales', 'HR']
})

print("Original DataFrame:")
print(df, "\n")

# 1️⃣ Filtering using loc: Age > 25 AND Score > 70
filtered_loc = df.loc[(df['Age'] > 25) & (df['Score'] > 70)]
print("Filtered using .loc (Age > 25 and Score > 70):")
print(filtered_loc, "\n")

# 2️⃣ Filtering using query(): Department is HR OR Score >= 90
filtered_query = df.query("Department == 'HR' or Score >= 90")
print("Filtered using .query(): (Department = HR OR Score >= 90)")
print(filtered_query)
```

Output:

```
Original DataFrame:
    Name  Age  Score Department
0  Alice   24     88         IT
1    Bob   30     75         HR
2  Carol   22     92         IT
3  David   28     60      Sales
4    Eva   35     85         HR

Filtered using .loc (Age > 25 and Score > 70):
  Name  Age  Score Department
1  Bob   30     75         HR
4  Eva   35     85         HR

Filtered using .query(): (Department = HR OR Score >= 90)
    Name  Age  Score Department
1    Bob   30     75         HR
2  Carol   22     92         IT
4    Eva   35     85         HR
```

Practical No. 6

AIM:

Using NumPy, create two matrices (3x3) filled with random integers. Perform matrix addition, subtraction, multiplication, element-wise division, and calculate the determinant and inverse.

Theory:

2D arrays (matrices) are essential in mathematical and scientific computations.
NumPy allows element-wise and matrix operations like addition, subtraction, multiplication, and division.
Determinant and inverse of matrices are important in solving systems of equations.
Matrix processing forms the foundation for AI, image processing, and engineering calculations.

Input (Example Code):

```
import numpy as np

# Creating two random 3x3 matrices (values between 1 and 10)
A = np.random.randint(1, 10, (3, 3))
B = np.random.randint(1, 10, (3, 3))

print("Matrix A:")
print(A, "\n")
print("Matrix B:")
print(B, "\n")

# Matrix Operations
add_result = A + B
sub_result = A - B
mul_result = np.dot(A, B)  # Matrix multiplication
div_result = A / B         # Element-wise division

print("Addition (A + B):")
print(add_result, "\n")
print("Subtraction (A - B):")
print(sub_result, "\n")
print("Multiplication (A x B):")
print(mul_result, "\n")
print("Element-wise Division (A / B):")
print(div_result, "\n")

# Determinant and Inverse of Matrix A
det_A = np.linalg.det(A)
inv_A = np.linalg.inv(A)

print("Determinant of A:", det_A, "\n")
print("Inverse of A:")
print(inv_A)
```

Output:

```
Matrix A:
[[9 1 3]
 [5 5 5]
 [6 5 3]]

Matrix B:
[[9 2 1]
 [3 6 9]
 [6 5 8]]

Addition (A + B):
[[18  3  4]
 [ 8 11 14]
 [12 10 11]]

Subtraction (A - B):
[[ 0 -1  2]
 [ 2 -1 -4]
 [ 0  0 -5]]

Multiplication (A x B):
[[102  39  42]
 [ 90  65  90]
 [ 87  57  75]]

Element-wise Division (A / B):
[[1.         0.5        3.        ]
 [1.66666667 0.83333333 0.55555556]
 [1.         1.         0.375     ]]

Determinant of A: -90.0

Inverse of A:
[[ 0.11111111 -0.13333333  0.11111111]
 [-0.16666667 -0.1         0.33333333]
 [ 0.05555556  0.43333333 -0.44444444]]
```

Practical No. 7

AIM:

Write a Pandas program to merge two DataFrames using different types of joins (inner, outer, left, right).

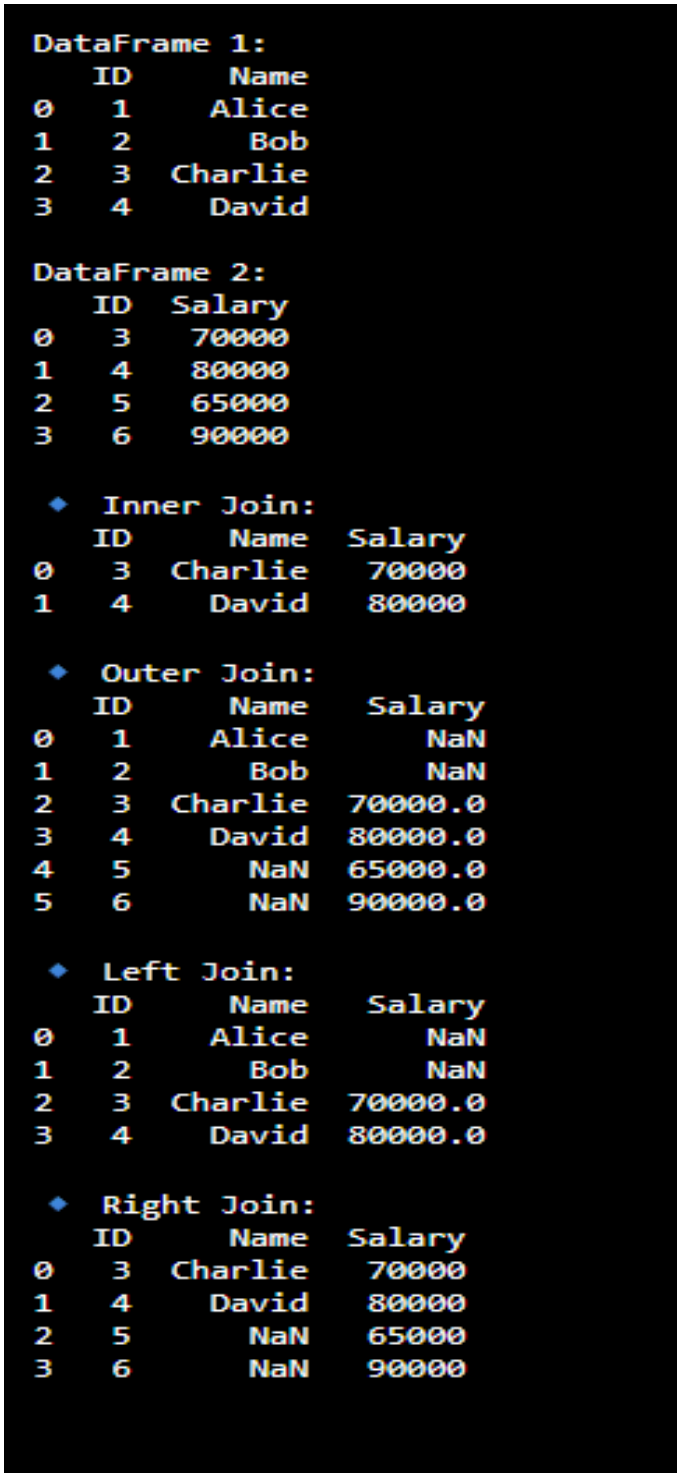
Theory:

Data is often stored in multiple tables, and merging helps combine them into one view.
Pandas supports joins like inner, outer, left, and right to control how data matches.
Keys or common columns are used to map related information between DataFrames.
Data merging is essential in data warehousing, business intelligence, and databases.

Input (Example Code):

```
import pandas as pd

# Sample DataFrames
df1 = pd.DataFrame({
    'ID': [1, 2, 3, 4],
    'Name': ['Alice', 'Bob', 'Charlie', 'David']
})

df2 = pd.DataFrame({
    'ID': [3, 4, 5, 6],
    'Salary': [70000, 80000, 65000, 90000]
})

print("DataFrame 1:")
print(df1, "\n")

print("DataFrame 2:")
print(df2, "\n")

# ⭐ Inner Join
inner_join = pd.merge(df1, df2, on='ID', how='inner')
print("🔹 Inner Join:")
print(inner_join, "\n")

# ⭐ Outer Join
outer_join = pd.merge(df1, df2, on='ID', how='outer')
print("🔹 Outer Join:")
print(outer_join, "\n")

# ⭐ Left Join
left_join = pd.merge(df1, df2, on='ID', how='left')
print("🔹 Left Join:")
print(left_join, "\n")

# ⭐ Right Join
right_join = pd.merge(df1, df2, on='ID', how='right')
print("🔹 Right Join:")
print(right_join)
```

Output:



```
DataFrame 1:
   ID     Name
0   1    Alice
1   2      Bob
2   3  Charlie
3   4    David

DataFrame 2:
   ID  Salary
0   3   70000
1   4   80000
2   5   65000
3   6   90000

 🔹 Inner Join:
   ID     Name  Salary
0   3  Charlie   70000
1   4    David   80000

 🔹 Outer Join:
   ID     Name   Salary
0   1    Alice      NaN
1   2      Bob      NaN
2   3  Charlie  70000.0
3   4    David  80000.0
4   5      NaN  65000.0
5   6      NaN  90000.0

 🔹 Left Join:
   ID     Name   Salary
0   1    Alice      NaN
1   2      Bob      NaN
2   3  Charlie  70000.0
3   4    David  80000.0

 🔹 Right Join:
   ID     Name  Salary
0   3  Charlie   70000
1   4    David   80000
2   5      NaN   65000
3   6      NaN   90000
```

Practical No. 8

AIM:

Load a dataset and perform time-series analysis using Pandas DateTime features.

Theory:

Time-series data is indexed by time and used to observe patterns and trends.
Pandas DateTime features help convert date columns and set them as indexes.
Resampling (daily, monthly, weekly) summarizes time-based data efficiently.
Rolling or moving averages help in forecasting and trend analysis.

Input (Example Code):

```
import pandas as pd

# Create a sample time-series dataset
data = {
    'date': ['2025-01-01', '2025-01-02', '2025-01-03', '2025-01-04', '2025-01-05'],
    'sales': [120, 150, 130, 170, 160]
}

df = pd.DataFrame(data)

print("Original DataFrame:")
print(df, "\n")

# Convert column to DateTime
df['date'] = pd.to_datetime(df['date'])

print("Data types after conversion:")
print(df.dtypes, "\n")

# Set Date as index (very useful for time-series)
df.set_index('date', inplace=True)
print("DataFrame with Date as index:")
print(df, "\n")

# 📌 Extract DateTime Features
df['year'] = df.index.year
df['month'] = df.index.month
df['day'] = df.index.day
df['weekday'] = df.index.day_name()

print("With extracted DateTime features:")
print(df, "\n")

# 📌 Resampling - Weekly Summary
weekly_summary = df['sales'].resample('W').sum()
print("Weekly Sales Summary:")
print(weekly_summary, "\n")

# 📌 Rolling Mean (Moving Average)
df['rolling_mean_3'] = df['sales'].rolling(window=3).mean()
print("With 3-day Rolling Mean:")
print(df)
```

Output:

```
Original DataFrame:
         date  sales
0  2025-01-01    120
1  2025-01-02    150
2  2025-01-03    130
3  2025-01-04    170
4  2025-01-05    160

Data types after conversion:
date     datetime64[ns]
sales             int64
dtype: object

DataFrame with Date as index:
            sales
date
2025-01-01    120
2025-01-02    150
2025-01-03    130
2025-01-04    170
2025-01-05    160

With extracted DateTime features:
            sales  year  month  day    weekday
date
2025-01-01    120  2025      1    1  Wednesday
2025-01-02    150  2025      1    2   Thursday
2025-01-03    130  2025      1    3     Friday
2025-01-04    170  2025      1    4   Saturday
2025-01-05    160  2025      1    5     Sunday

Weekly Sales Summary:
date
2025-01-05    730
Freq: W-SUN, Name: sales, dtype: int64

With 3-day Rolling Mean:
            sales  year  month  day    weekday  rolling_mean_3
date
2025-01-01    120  2025      1    1  Wednesday             NaN
2025-01-02    150  2025      1    2   Thursday             NaN
2025-01-03    130  2025      1    3     Friday      133.333333
2025-01-04    170  2025      1    4   Saturday      150.000000
2025-01-05    160  2025      1    5     Sunday      153.333333
```

Practical No. 9

AIM:

Using NumPy, generate a 1D array of 100 random integers between 1 and 1000. Use Boolean indexing to filter all values greater than 500 and less than 800, and calculate the mean of filtered values.

Theory:

Boolean indexing filters an array based on conditions like greater than or equal to.
It helps extract only meaningful values from large numeric datasets.
After filtering, NumPy statistical functions compute summaries like mean or max.
This improves performance over traditional loops and manual filtering.

Input (Example Code):

```
import numpy as np

# Generate 1D array with 100 random integers between 1 and 1000
arr = np.random.randint(1, 1001, size=100)

print("Original Array:")
print(arr, "\n")

# Boolean filtering: values > 500 and < 800
filtered_values = arr[(arr > 500) & (arr < 800)]

print("Filtered Values ( >500 and <800 ):")
print(filtered_values, "\n")

# Calculate mean of filtered values
mean_value = filtered_values.mean()

print("Mean of Filtered Values:")
print(mean_value)
```

Output:

```
Original Array:
[ 818  163   80   83    9   78  287  498  120  312  804  572  998  201
  181  381  919  608  385  917  651  772  447  427  997  462  775  102
  345  346  361    4  370  737   13  273  814  971  204  288   54  445
  875 1000  300  385  904   70  222  267  686  781  565  995  705  286
  672  339  939  684  196  376  787  350  695  377  283  818  319  689
  322  717  686  139   35  878  750  817  604  541  693  383  635  263
    9  565  514  339  410  248  670   88  969  207  547  200  359  639
  145  605]

Filtered Values ( >500 and <800 ):
[572 608 651 772 775 737 686 781 565 705 672 684 787 695 689 717 686 750
 604 541 693 635 565 514 670 547 639 605]

Mean of Filtered Values:
662.3214285714286
```

Practical No. 10

AIM:

Write a Pandas program to pivot a DataFrame to create a pivot table summarizing the data.

Theory:

Pivot tables convert detailed data into summarized reports.
Pandas `pivot_table()` groups data and applies aggregation functions like mean or sum.
It helps analyze relationships between multiple columns (e.g., department vs salary).
Widely used for MIS reports, business analytics, and data visualization.

Input (Example Code):

```
import pandas as pd

# Sample dataset
data = {
    'Department': ['IT', 'IT', 'HR', 'HR', 'Sales', 'Sales'],
    'Employee': ['Amit', 'Neha', 'Priya', 'Rohit', 'Karan', 'Sneha'],
    'Salary': [50000, 65000, 45000, 47000, 60000, 62000],
    'Bonus': [5000, 7000, 3000, 3500, 5500, 6000]
}

df = pd.DataFrame(data)

print("Original DataFrame:")
print(df, "\n")

# Creating Pivot Table -> Summarizing data by Department
pivot = df.pivot_table(
    values=['Salary', 'Bonus'],     # values to summarize
    index='Department',             # group by department
    aggfunc='mean'                  # calculating mean
)

print("Pivot Table (Department wise average Salary & Bonus):")
print(pivot)
```

Output:

```
Original DataFrame:
  Department Employee  Salary  Bonus
0         IT     Amit   50000   5000
1         IT     Neha   65000   7000
2         HR    Priya   45000   3000
3         HR    Rohit   47000   3500
4      Sales    Karan   60000   5500
5      Sales    Sneha   62000   6000

Pivot Table (Department wise average Salary & Bonus):
             Bonus   Salary
Department
HR          3250.0  46000.0
IT          6000.0  57500.0
Sales       5750.0  61000.0
```

--*-*-*-*-*