**Madiha Aimon Tappal**

[**madihaaimon@gmail.com**](mailto:madihaaimon@gmail.com)

**Data Engineering Batch – 1**

**Day – 10 Assignment**

**Python**

**Pandas**

Pandas is a powerful and popular open-source data manipulation and analysis library for Python. It provides easy-to-use data structures, such as Series and Data Frame, along with a wide variety of functions to manipulate and analyze structured data seamlessly.

**Key Components of Pandas:**

**Data Frame:**

* + A two-dimensional, labelled data structure resembling a table in a relational database.
  + Columns can be of different types (integers, strings, floats, etc.).
  + Allows for easy indexing, slicing, and reshaping of data.

import pandas as pd

df = pd.DataFrame(data={'Column1': [1, 2, 3], 'Column2': ['A', 'B', 'C']})

**Series:**

* A one-dimensional labeled array capable of holding any data type.
* Can be thought of as a single column or row of data within a Data Frame.

**Key Operations and Features:**

1. **Reading and Writing Data:**
   * **read\_csv()**, **read\_excel()**, and other methods for reading data from various file formats.
   * **to\_csv()**, **to\_excel()**, etc., for writing data back to different formats.
2. **Data Cleaning and Transformation:**
   * Handling missing data using methods like **dropna()** and **fillna()**.
   * Renaming columns using **rename()** method.
   * Merging and joining Data Frames using **merge()** method.
3. **Filtering and Selection:**
   * Boolean indexing and filtering data using conditions.
   * Selecting specific columns or rows using **loc[]** and **iloc[]**.
4. **Grouping and Aggregation:**
   * **Groupby()** method for grouping data based on one or more columns.
   * Aggregation functions like **sum()**, **mean()**, and **count()** to summarize grouped data.
5. **Time Series Data:**
   * Specialized tools for handling time series data, including date and time-based indexing.
   * Resampling and time-based operations.

df['Date'] = pd.to\_datetime(df['Date'])

df.set\_index('Date', inplace=True)

**Plotting:**

* Integration with Matplotlib for easy plotting of data using **plot()** method.

df.plot(x='Date', y='Value', kind='line')

**Statistical Analysis:**

* Descriptive statistics like **mean()**, **std()**, and **describe()**.
* Correlation and covariance calculations.

correlation\_matrix = df.corr()

**Use Cases:**

* Data cleaning and preprocessing.
* Exploratory data analysis.
* Time series analysis.
* Merging and joining datasets.
* Handling missing data and outliers.

**Reading CSV Data using Pandas:**

Reading CSV data using Pandas is a common operation in data analysis and manipulation. Pandas provides the **read\_csv** function to read data from CSV files and create a Data Frame. Here's a detailed explanation:

**Theory:**

CSV (Comma-Separated Values) is a popular file format for storing tabular data. The **read\_csv** function in Pandas is used to read data from CSV files into a Pandas Data Frame. The DataFrame is a two-dimensional, labelled data structure with columns that can be of different types (integers, strings, etc.). This function automatically detects the delimiter and other parameters to load the data correctly.

import pandas as pd

df = pd.read\_csv('your\_data.csv')

**CSV (Comma-Separated Values):**

* CSV is a plain text format for tabular data where each line represents a row, and columns are separated by a delimiter (commonly a comma).
* It is a widely used format for storing and exchanging structured data.

**Pandas read\_csv Function:**

* **read\_csv** is a Pandas function used to read data from CSV files into a Data Frame.
* It automatically infers the delimiter, handles missing values, and supports various options for customization.

### Example:

Suppose you have a CSV file named **data.csv** with the following content:

Name, Age, City

Alice, 25, New York

Bob, 30, San Francisco

Charlie, 22, Los Angeles

You can read this CSV file into a Pandas Data Frame using the **read\_csv** function:

import pandas as pd

# Reading CSV data into a DataFrame

df = pd.read\_csv('data.csv')

# Displaying the DataFrame

print(df)

The resulting Data Frame **df** will look like:

Name Age City

0 Alice 25 New York

1 Bob 30 San Francisco

2 Charlie 22 Los Angeles

**Filter Data in Pandas Data frame using query:**

Filtering data in a Pandas DataFrame using the **query** method allows you to extract subsets of data based on specific conditions.

**Theory:**

**Data Filtering:**

* Data filtering is the process of selecting a subset of data based on certain conditions or criteria.
* In Pandas, you can filter data using boolean indexing or the more expressive **query** method.

**Pandas query Method:**

* The **query** method in Pandas provides a concise and SQL-like syntax for filtering data.
* It allows you to write conditions directly as strings, making it more readable and expressive.

Filtering allows you to extract specific subsets of data based on conditions. The **query** method in Pandas provides a concise way to express these conditions using a SQL-like syntax.

filtered\_data = df.query('Status == "Completed"')

**Example:**

Suppose you have a DataFrame **df**:

import pandas as pd

data = {'Name': ['Alice', 'Bob', 'Charlie'],

'Age': [25, 30, 22],

'City': ['New York', 'San Francisco', 'Los Angeles']}

df = pd.DataFrame(data)

You can use the **query** method to filter the DataFrame based on a condition, for example, selecting rows where the age is greater than 25:

# Filtering data using the query method

filtered\_data = df.query('Age > 25')

# Displaying the filtered DataFrame

print(filtered\_data)

The resulting **filtered\_data** DataFrame will look like:

Name Age City

1 Bob 30 San Francisco

### Get Count by Status using Pandas Data frame APIs:

Getting the count of unique values in a specific column, such as "Status," in a Pandas DataFrame is a common operation. The **value\_counts()** method is particularly useful for this task. Here's an explanation:

**Theory:**

**Counting Unique Values:**

* Understanding the distribution of categorical data, like the different statuses in a dataset, is crucial for data analysis.
* The **value\_counts()** method in Pandas is used to obtain a Series containing counts of unique values.

**Example:**

Suppose you have a DataFrame **df**:

import pandas as pd

data = {'Name': ['Alice', 'Bob', 'Charlie', 'David', 'Emily'],

'Status': ['Completed', 'In Progress', 'Completed', 'In Progress', 'Completed']}

df = pd.DataFrame(data)

You can use **value\_counts()** to get the count of each unique status:

# Getting count by Status

status\_count = df['Status'].value\_counts()

# Displaying the count by Status

print(status\_count)

The output will look like:

Completed 2

In Progress 2

Name: Status, dtype: int64

The **value\_counts()** method is a quick and effective way to understand the distribution of categorical data in a specific column of a Pandas DataFrame. It's particularly useful for tasks like counting the occurrences of different statuses, as shown in the example.

Top of Form

### Get count by Month and Status using Pandas Data frame APIs:

To get the count by both Month and Status in a Pandas DataFrame, you can use the **groupby** method along with additional aggregation functions. Here's an explanation:

**Theory:**

**Grouping and Aggregating:**

* **groupby** is a powerful method in Pandas that allows you to group rows based on one or more columns.
* After grouping, you can use aggregation functions (such as **size()**, **count()**, **sum()**, etc.) to perform operations on each group.

**Example:**

Suppose you have a DataFrame **df** with a 'Date' column:

import pandas as pd

data = {'Date': ['2022-01-01', '2022-01-01', '2022-02-01', '2022-02-01', '2022-02-01'],

'Status': ['Completed', 'In Progress', 'Completed', 'In Progress', 'Completed']}

df = pd.DataFrame(data)

# Convert 'Date' to datetime format

df['Date'] = pd.to\_datetime(df['Date'])

You can get the count by both Month and Status using the following code:

# Extract Month from 'Date'

df['Month'] = df['Date'].dt.month

# Group by 'Month' and 'Status', and get count for each group

count\_by\_month\_status = df.groupby(['Month', 'Status']).size().unstack()

# Display the result

print(count\_by\_month\_status)

The output will look like:

Status Completed In Progress

Month

1 1 1

2 2 1

**Explanation:**

1. **Extracting Month:**
   * The **dt.month** attribute of the datetime column is used to extract the month and create a new 'Month' column.
2. **Grouping and Aggregating:**
   * The **groupby(['Month', 'Status'])** groups the data by both 'Month' and 'Status'.
   * The **size()** function counts the number of occurrences in each group.
   * **unstack()** is used to pivot the resulting Series into a DataFrame.
3. **Displaying the Result:**
   * The final DataFrame shows the count of each status for each month.

The **groupby** functionality in Pandas is versatile and can be adapted to various scenarios, allowing you to analyze and summarize your data effectively.

### Create Data frames using dynamic column list on CSV Data:

Creating Data Frames using a dynamic column list allows you to select and load only specific columns from a CSV file. This is useful when you don't need all the columns in your analysis, improving efficiency and reducing memory usage. Here's an explanation:

**Theory:**

**Dynamic Column Selection:**

* The **usecols** parameter in the **read\_csv** function allows you to specify a subset of columns to load from a CSV file.
* By providing a dynamic list of column names, you can control which columns are included in the resulting DataFrame.

Sometimes, you may only need a subset of columns from a CSV file. The **usecols** parameter in **read\_csv** allows you to specify which columns to load into the DataFrame, saving memory and improving performance.

**Example:**

Suppose you have a CSV file named **your\_data.csv** with columns 'Column1', 'Column2', 'Column3', and 'Column4'. You want to create a DataFrame by dynamically selecting columns based on a list of column names:

import pandas as pd

# Dynamic column list

dynamic\_columns = ['Column1', 'Column3']

# Creating a DataFrame with dynamic column selection

dynamic\_df = pd.read\_csv('your\_data.csv', usecols=dynamic\_columns)

# Displaying the DataFrame

print(dynamic\_df)

**Explanation:**

1. **Dynamic Column List:**
   * **dynamic\_columns** is a list containing the column names you want to include in the DataFrame dynamically.
2. **Reading CSV with Dynamic Column Selection:**
   * The **read\_csv** function reads the CSV file ('your\_data.csv') and uses the **usecols** parameter to select only the columns specified in the dynamic column list.
3. **Displaying the Result:**
   * The resulting DataFrame (**dynamic\_df**) will only contain the columns specified in the **dynamic\_columns** list.

### Performing Inner Join between Pandas Dataframes:

Performing an inner join between Pandas DataFrames involves combining rows from two DataFrames based on a common key, similar to the concept of INNER JOIN in SQL. Here's an explanation:

**Theory:**

**Inner Join:**

* An inner join returns only the rows where there is a match in both DataFrames based on a specified key.
* It's a way to combine information from two tables by aligning rows with matching keys.

**Pandas merge Function:**

* The **merge** function in Pandas is used to perform various types of joins, including inner joins.
* It allows you to specify the key(s) on which the DataFrames should be joined, as well as the type of join (inner, outer, left, right).

**Example:**

Suppose you have two DataFrames, **df1** and **df2**, with a common key 'ID':

import pandas as pd

data1 = {'ID': [1, 2, 3], 'Value1': ['A', 'B', 'C']}

data2 = {'ID': [2, 3, 4], 'Value2': ['X', 'Y', 'Z']}

df1 = pd.DataFrame(data1)

df2 = pd.DataFrame(data2)

You can perform an inner join using the **merge** function:

# Performing inner join on 'ID'

merged\_df = pd.merge(df1, df2, on='ID', how='inner')

# Displaying the result

print(merged\_df)

**Explanation:**

1. **Common Key:**
   * The 'ID' column is the common key between the two Data Frames. This column is used to match rows during the join.
2. **Inner Join Using merge:**
   * The **merge** function is called on one of the Data Frames (**df1**), and the other DataFrame (**df2**) is specified as well as the common key ('ID').
   * The **how='inner'** parameter specifies that it's an inner join.
3. **Displaying the Result:**
   * The resulting **merged\_df** DataFrame contains only the rows where the 'ID' values match in both Data Frames, and columns from both Data Frames are combined.

### Performing Aggregations on Join Results:

After performing a join between Pandas DataFrames, you might want to aggregate the results based on certain criteria. Here's an example of how to perform aggregations on join results:

import pandas as pd

# Sample DataFrames

data1 = {'ID': [1, 2, 3], 'Value1': [10, 20, 30]}

data2 = {'ID': [2, 3, 4], 'Value2': [40, 50, 60]}

df1 = pd.DataFrame(data1)

df2 = pd.DataFrame(data2)

# Performing inner join

merged\_df = pd.merge(df1, df2, on='ID', how='inner')

# Aggregating the results based on 'ID'

aggregated\_result = merged\_df.groupby('ID').agg({'Value1': 'sum', 'Value2': 'mean'})

# Displaying the aggregated result

print(aggregated\_result)

In this example, after performing an inner join on 'ID', we use the **groupby** method to group the results by 'ID' and then apply aggregation functions (**sum** for 'Value1' and **mean** for 'Value2')

### Sort Data in Pandas Data Frames:

Sorting data in a Pandas DataFrame can be done using the **sort\_values** method. Here's an example:

import pandas as pd

# Sample DataFrame

data = {'ID': [3, 1, 2], 'Value': [30, 10, 20]}

df = pd.DataFrame(data)

# Sorting DataFrame by 'ID'

sorted\_df = df.sort\_values(by='ID', ascending=True)

# Displaying the sorted DataFrame

print(sorted\_df)

In this example, the **sort\_values** method is used to sort the DataFrame based on the 'ID' column in ascending order.

### Writing Pandas DataFrames to Files:

You can write Pandas DataFrames to various file formats using methods like **to\_csv** or **to\_excel**. Here's an example of writing a DataFrame to a CSV file:

import pandas as pd

# Sample DataFrame

data = {'ID': [1, 2, 3], 'Value': [10, 20, 30]}

df = pd.DataFrame(data)

# Writing DataFrame to CSV file

df.to\_csv('output\_data.csv', index=False)

In this example, the **to\_csv** method is used to write the DataFrame to a CSV file named 'output\_data.csv'. The **index=False** parameter avoids writing row indices to the file.

### Write Pandas DataFrames to JSON Files:

To write a Pandas DataFrame to a JSON file, you can use the **to\_json** method:

import pandas as pd

# Sample DataFrame

data = {'ID': [1, 2, 3], 'Value': [10, 20, 30]}

df = pd.DataFrame(data)

# Writing DataFrame to JSON file

df.to\_json('output\_data.json', orient='records')

In this example, the **to\_json** method is used with **orient='records'** to write the DataFrame to a JSON file named 'output\_data.json'. The **orient** parameter specifies the format of the JSON file.