**Madiha Aimon Tappal**

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

**Data Engineering Batch – 1**

**Coding Challenge**

**Python**

* **Pandas for Data Processing:**

Pandas is a powerful data manipulation and analysis library for Python. It provides data structures like Series and DataFrame, which are designed for efficiently manipulating large datasets.

**1. Data Structures:**

* **Series:** A one-dimensional labelled array capable of holding any data type. It is essentially a column in a Pandas DataFrame.
* import pandas as pd
* # Creating a Series

s = pd.Series([1, 3, 5, np.nan, 6, 8])

* **DataFrame:** A two-dimensional labelled data structure with columns that can be of different types. It is similar to a spreadsheet or SQL table.
* # Creating a DataFrame

df = pd.DataFrame({'A': 1., 'B': pd.Timestamp('20220101'), 'C': pd.Series(1, index=list(range(4)), dtype='float32'), 'D': np.array([3] \* 4, dtype='int32'), 'E': pd.Categorical(["test", "train", "test", "train"]), 'F': 'foo'})

**2. Operations on DataFrames:**

* **Data Cleaning:** Pandas facilitates handling missing data through methods like **dropna()** or **fillna()**.
* # Dropping missing values
* df.dropna()
* # Filling missing values

df.fillna(value=5)

* **Data Transformation:** Allows reshaping, merging, and pivoting of data using methods such as **merge()**, **pivot\_table()**, and **melt()**.
* # Merging DataFrames
* pd.merge(df1, df2, on='key')
* # Pivoting DataFrames

df.pivot\_table(index='Date', columns='Category', values='Value', aggfunc='sum')

* **Data Aggregation:** Supports grouping and aggregation operations using **groupby()** and aggregation functions.
* # Grouping by a column and calculating mean

df.groupby('Category').mean()

**3. Indexing and Selection:**

* **Label-Based Indexing:** Enables accessing data using labels with **loc[]**.

df.loc[:, ['A', 'B']]

* **Positional Indexing:** Allows accessing data by integer location with **iloc[]**.

df.iloc[3]

* **Boolean Indexing:** Permits filtering data based on conditions.

df[df['A'] > 0]

**4. Time Series Handling:**

* Pandas is equipped with tools for working with time-series data, allowing for time-based indexing and resampling.

# Creating a time series DataFrame

time\_series = pd.DataFrame({'value': [1, 2, 3, 4]}, index=pd.to\_datetime(['20220101', '20220102', '20220103', '20220104']))

# Resampling time series data

time\_series.resample('D').sum()

#### **Advantages:**

**1. Flexibility:**

* Pandas provides a flexible and high-performance set of data structures, making it easy to work with various types of data.

**2. Data Alignment and Handling:**

* Automatically aligns data based on labels, simplifying operations on different datasets.

**3. Integration with Other Libraries:**

* Integrates seamlessly with other libraries, such as NumPy, Matplotlib, and scikit-learn, enhancing its capabilities.

**4. Efficient Data Storage:**

* Efficiently stores large datasets and handles various file formats for input and output.

**5. Rich Functionality:**

* Offers a rich set of functions for data manipulation, exploration, and analysis, reducing the need for extensive coding.

**6. Wide Adoption:**

* Pandas is widely adopted in both industry and academia, leading to a large and active community, extensive documentation, and numerous online resources.

**7. Ease of Learning:**

* Has a relatively gentle learning curve, making it accessible to beginners while providing advanced features for experienced users.

Pandas is a powerful tool for data processing, and its ability to handle diverse tasks efficiently makes it an essential library in the Python data science ecosystem.

### Reading CSV Data using Pandas:

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

* CSV is a plain-text format commonly used for representing tabular data. In a CSV file, each line represents a row of the table, and columns are separated by a delimiter (usually a comma, but it can be a different character).

**2. Pandas read\_csv Function:**

* The **read\_csv** function in Pandas is used to read data from a CSV file into a Pandas DataFrame. It is a powerful and flexible tool for handling various CSV formats.

**3. Parameters of read\_csv:**

* **filepath\_or\_buffer**: The path to the CSV file or a URL. It can also be a file-like object.
* **sep**: The delimiter used in the CSV file. By default, it is a comma (**,**), but it can be changed based on the actual delimiter in the file.
* **header**: Specifies which row to use as column names. By default, it uses the first row.
* **index\_col**: Specifies which column to use as the DataFrame index.
* **encoding**: Specifies the character encoding to be used to interpret the file.

**4. DataFrame:**

* The DataFrame is a two-dimensional labeled data structure in Pandas, resembling a table or a spreadsheet. It has rows and columns, and each column can have a different data type.

#### **Advantages:**

**1. Flexibility:**

* **read\_csv** can handle a wide variety of CSV formats, making it flexible for different types of datasets.

**2. Automatic Type Inference:**

* Pandas automatically infers the data types of columns based on the content of the CSV file, reducing the need for manual type specification.

**3. Efficient Handling of Large Datasets:**

* Pandas efficiently loads and handles large datasets, providing tools for memory optimization and selective loading of columns.

**4. Handling Missing Values:**

* **read\_csv** provides options to handle missing values, such as specifying a placeholder for missing values or automatically recognizing common representations of missing data.

**5. Customization:**

* It allows customization of various parameters, such as specifying which row to use as headers, the delimiter used, and how to handle quoting.

#### **Example:**

Assuming a CSV file named **example.csv** with the following content:

Name,Age,Occupation

John,30,Engineer

Alice,25,Doctor

Bob,35,Teacher

You can read this CSV file into a Pandas DataFrame using the following code:

import pandas as pd

# Replace 'example.csv' with the actual path to your CSV file

file\_path = 'example.csv'

# Read CSV data into a DataFrame

df = pd.read\_csv(file\_path)

# Display the DataFrame

print(df)

This would output:

Name Age Occupation

0 John 30 Engineer

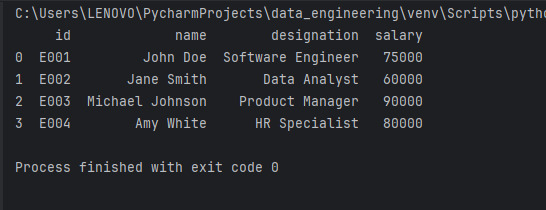
1 Alice 25 Doctor

2 Bob 35 Teacher

**Example: -**

import pandas as pd  
  
file\_path = 'new\_file.csv'  
  
# Read CSV file into a DataFrame  
df = pd.read\_csv(file\_path)  
  
# Display the first few rows of the DataFrame  
print(df.head())

Output:



### Reading Data from CSV Files to Pandas DataFrames

#### **1. CSV (Comma-Separated Values):**

* CSV is a common plain-text format used for storing tabular data. Each line in the file typically represents a row, and values in each row are separated by a delimiter, usually a comma.

#### **2. Pandas** read\_csv **Function:**

* **read\_csv** is a powerful function in the Pandas library designed to read data from CSV files into a Pandas DataFrame. It offers a range of options to handle different CSV file structures.

import pandas as pd

# Read CSV data into a DataFrame

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

#### **3. Parameters of** read\_csv**:**

* **filepath\_or\_buffer**: The path to the CSV file or a URL. It can also be a file-like object.
* **sep**: The delimiter used in the CSV file (default is a comma **,**).
* **header**: Specifies which row to use as column names. By default, it uses the first row.
* **index\_col**: Specifies which column to use as the DataFrame index.
* **encoding**: Specifies the character encoding to be used to interpret the file.

# Example with additional parameters

df = pd.read\_csv('your\_file.csv', sep='\t', header=0, index\_col='ID', encoding='utf-8')

#### **4. DataFrame:**

* A DataFrame is a two-dimensional labeled data structure in Pandas. It consists of rows and columns, where each column can have a different data type. It is a powerful and flexible tool for data manipulation and analysis.

import pandas as pd

# Creating a DataFrame from a dictionary

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

'Age': [30, 25, 35],

'Occupation': ['Engineer', 'Doctor', 'Teacher']}

df = pd.DataFrame(data)

#### **. Advantages:**

* **Flexibility:** **read\_csv** can handle various CSV formats, making it flexible for different types of datasets.
* **Automatic Type Inference:** Pandas infers data types, reducing the need for manual specification.
* **Efficient Handling of Large Datasets:** Pandas efficiently loads and handles large datasets, providing tools for memory optimization.
* **Handling Missing Values:** **read\_csv** provides options to handle missing values, such as specifying a placeholder or automatically recognizing common representations.
* **Customization:** It allows customization of parameters such as headers, delimiters, and encoding.

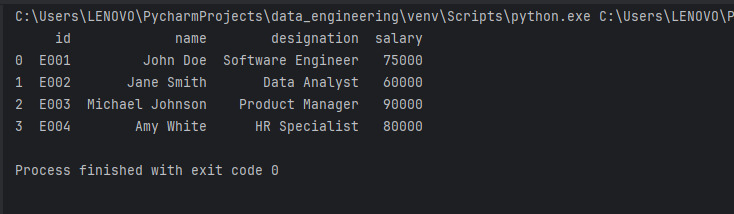
#### **6. Example:**

* Reading data from a CSV file into a Pandas DataFrame is as simple as using the **read\_csv** function, as demonstrated in the provided Python code.

Example:

import pandas as pd  
  
file\_path = 'new\_file.csv'  
  
df = pd.read\_csv(file\_path)  
  
# Read CSV file into a DataFrame  
data\_frame = pd.DataFrame(df)  
  
print(data\_frame)

Output:



### Filtering Data in Pandas DataFrame using query

#### **1. Introduction to Data Filtering in Pandas:**

* Filtering data is a common operation in data analysis, allowing you to extract specific rows or columns based on certain conditions.
* Pandas provides the **query** method, which enables you to filter DataFrame rows using a query expression.

#### **2. The** query **Method:**

* The **query** method is a convenient and expressive way to filter rows in a DataFrame based on a query expression.
* It takes a string representing a boolean expression and returns a DataFrame containing only the rows that satisfy the condition.

# Example of using the query method

filtered\_df = df.query('Age > 25 and Occupation == "Engineer"')

#### **3. Query Expression Syntax:**

* The query expression is a string that resembles a SQL WHERE clause.
* You can reference column names directly in the expression.
* Logical operators such as **and**, **or**, and **not** can be used.
* Comparison operators like **==**, **<**, **>**, **<=**, and **>=** are used for conditional filtering.

#### **4. Example Query Expressions:**

* **Numeric Comparison:**

df.query('Age > 25')

* **String Comparison:**

df.query('Occupation == "Engineer"')

* **Combining Conditions:**

df.query('Age > 25 and Occupation == "Engineer"')

#### **5. Benefits of Using** query**:**

* **Conciseness:** **query** allows for concise and expressive code for filtering, reducing the need for long, nested conditional statements.
* **Readability:** Query expressions often resemble SQL, making the code more readable, especially for those familiar with SQL syntax.
* **Performance:** For large datasets, using **query** can be more performant than traditional boolean indexing.

#### **Example:**

* Suppose you have a DataFrame **df** with columns 'Age' and 'Occupation'. You can filter rows where Age is greater than 25 and Occupation is 'Engineer' as follows:

filtered\_df = df.query('Age > 25 and Occupation == "Engineer"')

Example:

import pandas as pd  
  
# Creating a sample DataFrame  
data = {'Name': ['Alice', 'Bob', 'Charlie', 'David'],  
 'Age': [25, 30, 35, 28],  
 'City': ['New York', 'San Francisco', 'Los Angeles', 'Chicago']}  
df = pd.DataFrame(data)  
  
# Display the original DataFrame  
print("Original DataFrame:")  
print(df)  
  
# Using query to filter data (e.g., selecting individuals older than 30)  
filtered\_df = df.query('Age > 30')  
  
# Display the filtered DataFrame  
print("\nFiltered DataFrame:")  
print(filtered\_df)

Output:

