What are Pandas?

- Pandas is a Python library built on top of NumPy, designed for data manipulation and analysis.
- It provides powerful tools for working with structured data (e.g., tabular data, time series)

Key Features:

- Data Structures: Two primary data structures – Series (1D labeled array) and DataFrame (2D labeled table).
- Data Manipulation: Easy handling of missing data, filtering, and transformations.
- Flexible Indexing: Label-based and integer-based indexing.
- File I/O: Read and write operations for multiple formats like CSV, Excel, SQL, and JSON.
- Data Cleaning: Handling duplicates, missing values, and more.
- Powerful Operations: Grouping, merging, reshaping, and aggregating data.

Installing Pandas

The first step in working with Pandas is to ensure whether it is installed in the system or not. If not, then we need to install it on our system using the pip command.

Follow these steps to install Pandas:

Step 1: Type ‘cmd’ in the search box and open it.
Step 2: Locate the folder using the cd command where the python-pip file has been installed.
Step 3: After locating it, type the command:

```
pip install pandas
```

install Pandas in PyCharm

Step 1: Open PyCharm

- Launch your PyCharm IDE.

Step 2: Open Your Project

- Select the project where you want to install Pandas.

Step 3: Access Project Settings

1. Click on File in the menu bar.
2. Select Settings (Windows/Linux) or Preferences (Mac).

Step 4: Navigate to Python Interpreter

1. In the settings window, go to Project: [Your Project Name] > Python Interpreter.
2. You'll see the list of installed packages for the selected Python environment.

Step 5: Install Pandas

1. Click the + button (usually at the top right of the interpreter window).
2. In the search bar, type pandas.
3. Select pandas from the results and click Install Package.

Step 6: Verify Installation

- Once installed, check that Pandas appears in the list of packages.
- Optionally, verify by running:

```python
import pandas as pd
print(pd.__version__)
```

Data Structures in Pandas Library

- Series
- DataFrame

1. Pandas Series:
   One-Dimensional: It is similar to a list or a 1D NumPy array.
   Labeled: It has an index (labels) for each element, which makes it more flexible than a regular list.
   Heterogeneous: It can hold different types of data (e.g., integers, strings, floats) in the same Series.

   Creating a Series:

   From a List:

   ```
   import pandas as pd

   data = [10, 20, 30, 40, 50]
   series = pd.Series(data)
   print(series)
   ```

   From a Dictionary:

```
data = {'a': 10, 'b': 20, 'c': 30}
series = pd.Series(data)
print(series)
```

From a NumPy Array:

```
import numpy as np
data = np.array([1.5, 2.5, 3.5, 4.5])
series = pd.Series(data)
print(series)
```

2. Pandas DataFrame:
    1. Two-Dimensional: It has both row and column labels (axes).
    2. Heterogeneous Data: Each column can contain different data types (e.g., integers, floats, strings).
    3. Size-Mutable: You can add, remove, or modify columns and rows.
    4. Supports Operations: You can perform vectorized operations (similar to NumPy) on DataFrames.

Creating a DataFrame:

1. From a Dictionary:

You can create a DataFrame by passing a dictionary where the keys become column names and values become column data.

```
import pandas as pd

data = {

    'Name': ['Alice', 'Bob', 'Charlie'],

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

    'City': ['New York', 'Los Angeles', 'Chicago']

}


df = pd.DataFrame(data)

print(df)
```

2. From a List of Lists:

You can create a DataFrame from a list of lists (or tuples), where each sublist represents a row, and you can specify column names.

```
data = [
    ['Alice', 25, 'New York'],
    ['Bob', 30, 'Los Angeles'],
    ['Charlie', 35, 'Chicago']
]


df = pd.DataFrame(data, columns=['Name', 'Age', 'City'])
print(df)
```

3. From a CSV File:

You can read data from a CSV file into a DataFrame using pd.read_csv().

```
df = pd.read_csv('data.csv')
print(df)
```

- **Indexing in Series:**

A Series is a one-dimensional labeled array. You can access elements by their index labels or integer positions.

```
import pandas as pd

# Creating a simple Series
data = pd.Series([10, 20, 30, 40], index=['a', 'b', 'c', 'd'])

# Accessing elements by index labels
print(data['a'])  # Output: 10

# Accessing elements by integer position
print(data[0])  # Output: 10
```

- **Indexing in DataFrame:**

A DataFrame is a two-dimensional labeled data structure with rows and columns. You can use both column labels and row labels for indexing.

```
# Creating a DataFrame
data = {
    'Name': ['Alice', 'Bob', 'Charlie'],
    'Age': [25, 30, 35],
    'City': ['New York', 'Los Angeles', 'Chicago']
}
df = pd.DataFrame(data)

# Accessing a column by label
print(df['Age'])

# Accessing a row by label using .loc
print(df.loc[1])  # Row with index 1 (second row)
```

```
# Accessing a row by integer position using .iloc
print(df.iloc[1])  # Row at position 1 (second row)
```

Exploring data in pandas

1) **Shape:** df.shape returns a tuple representing the dimensions of the DataFrame df. The tuple contains two elements: the number of rows and the number of columns in the DataFrame.
2) **Columns:** df.columns returns an Index object containing the column labels of the DataFrame df. It provides the names of all the columns in the DataFrame.
3) **Sample:** The sample() method is useful when you want to randomly select a subset of rows from a DataFrame.
4) **head()**
5) **tail()**
6) **describe()**: returns *statistical information about numerical column*
7) **info()**:

   df.info() provides a concise summary of the DataFrame df. It displays information about the DataFrame, including:

   - The number of rows (entries)
   - The number of columns
   - The column names and their data types
   - The number of non-null values in each column
   - The memory usage of the DataFrame

8) **isnull()**

9) **isnull().sum()**
10) **df.dtypes** returns a series containing the data type of each column in the DataFrame.
11) **df.dtypes == 'object'** compares each column's data type against the string 'object' and returns a boolean mask indicating which columns have the 'object' data type.
12) .index is used to extract the index labels (column names) of the columns that satisfy the condition.

```
import pandas as pb

#data=pb.read_csv("Orders.csv")

data1=pb.read_excel("EMPSAL (3).XLSX")

print(data1)

#print(data.head())#return starting 5 rows

#print(data.head(4))

#print(data.sample())#randome any 1 ros
return

#print(data.sample(3))#radomely rows are
getting return according limit

#print(data.isnull())#return result in
true and false format if null value-->true

#print(data.isnull().sum())

#print(data.shape)#return the shape of
your datafarme

#print(data.columns)

print(data1.describe().all)
```

Data aggregation :

List of Common Aggregation Functions

Function	Description
sum()	Calculates the sum of values.
mean()	Calculates the average (mean) of values.
median()	Calculates the median of values.
min()	Finds the minimum value.
max()	Finds the maximum value.
count()	Counts the number of non-NA/null values.
std()	Calculates the standard deviation.
var()	Calculates the variance.
prod()	Calculates the product of all values.
quantile()	Calculates the value at a given quantile.
mode()	Finds the mode(s) of values.
skew()	Measures the skewness of the distribution.
kurt()	Measures the kurtosis of the distribution.
cumsum()	Cumulative sum of values.
cumprod()	Cumulative product of values.
cummax()	Cumulative maximum of values.
cummin()	Cumulative minimum of values.

Data cleaning:

1. Handling Missing Values

Missing values can be handled by either filling them with a specific value or removing them entirely, depending on the context.

(a) Detecting Missing Values

Pandas uses NaN to represent missing data.

```
import pandas as pd

import numpy as np


# Sample DataFrame with missing values

data = {

    'Name': ['Alice', 'Bob', 'Charlie',
np.nan],

    'Age': [25, 30, np.nan, 22],

    'Salary': [50000, 60000, np.nan,
40000]
```

```
    }

    df = pd.DataFrame(data)


# Detect missing values

print(df.isnull())

print("Number of missing values:\n",
df.isnull().sum())
```

(b) Filling Missing Values

Use fillna to fill missing values. With parameters(ffill,bfill,how)

```
    # Fill missing values with a constant

    df['Age'] = df['Age'].fillna(0)


    # Fill missing values with the mean or
    median

    df['Salary'] =
    df['Salary'].fillna(df['Salary'].mean())


    print("After filling missing values:\n",
    df)
```

(c) Dropping Missing Values

Use dropna to remove rows or columns with missing values.

Example:

```
# Drop rows with any missing value

df_dropped_rows = df.dropna()


# Drop columns with any missing value

df_dropped_cols = df.dropna(axis=1)


print("After dropping rows:\n",
df_dropped_rows)

print("After dropping columns:\n",
df_dropped_cols)
```

2. Handling Duplicates

Duplicates can inflate data size and skew analysis. Use drop_duplicates to remove them.

```
    # Sample DataFrame with duplicates

    data = {
```

```
    'Name': ['Alice', 'Bob', 'Charlie',
'Alice'],

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

    'Salary': [50000, 60000, 40000, 50000]

}

df = pd.DataFrame(data)


# Check for duplicates

print("Duplicates:\n", df.duplicated())


# Remove duplicates

df_no_duplicates = df.drop_duplicates()


print("After removing duplicates:\n",
df_no_duplicates)
```

3. Renaming Columns

Column names may need to be standardized or renamed for clarity.

(a) Renaming Specific Columns

Use rename with a dictionary.

Example:

```
# Rename specific columns

df_renamed = df.rename(columns={'Name':
'Employee Name', 'Age': 'Employee Age'})


print("After renaming columns:\n",
df_renamed)
```

(b) Renaming All Columns

Use a list for all column names.

```
# Rename all columns

df.columns = ['Employee_Name',
'Employee_Age', 'Employee_Salary']


print("After renaming all columns:\n", df)
```

Data manipulation: groupby, merging, joining, and concatenation

1. Groupby

The groupby operation in Pandas allows you to group your data based on certain criteria and then perform aggregate functions on those groups (e.g., sum, mean, count).

Example:

```
import pandas as pd


df = pd.DataFrame({
    'Category': ['A', 'B', 'A', 'B', 'A'],
    'Value': [10, 20, 30, 40, 50]
})


grouped = df.groupby('Category')
print(grouped['Value'].sum())
```

Data manipulation techniques like groupby, merging, joining, and concatenation are essential for working with data in libraries like Pandas. Here's an overview of each:

1. Groupby

The groupby operation in Pandas allows you to group your data based on certain criteria and then perform aggregate functions on those groups (e.g., sum, mean, count).

Syntax:

python

Copy code

```
import pandas as pd


df = pd.DataFrame({
    'Category': ['A', 'B', 'A', 'B', 'A'],
    'Value': [10, 20, 30, 40, 50]
})


grouped = df.groupby('Category')
print(grouped['Value'].sum())
```

2. Merging

Merging combines two DataFrames based on a common column or index. It's similar to SQL joins (INNER, LEFT, RIGHT, OUTER).

Example:

```
df1 = pd.DataFrame({
    'ID': [1, 2, 3],
    'Name': ['Alice', 'Bob', 'Charlie']
})


df2 = pd.DataFrame({
    'ID': [1, 2, 4],
    'Age': [25, 30, 22]
})

merged_df = pd.merge(df1, df2, on='ID', how='inner')
print(merged_df)
```

3. Joining

Joining is a simplified version of merging, where the DataFrames are joined based on the index. It can be done using join()

Example:

```
df1 = pd.DataFrame({
    'Name': ['Alice', 'Bob', 'Charlie'],
    'Age': [25, 30, 22]
}, index=[1, 2, 3])

df2 = pd.DataFrame({
    'City': ['New York', 'San Francisco', 'Los Angeles']
}, index=[1, 2, 4])

joined_df = df1.join(df2)
print(joined_df)
```

4. Concatenation

Concatenation combines multiple DataFrames along a particular axis (rows or columns)

Example:

```
df1 = pd.DataFrame({
    'Name': ['Alice', 'Bob'],
    'Age': [25, 30]
})

df2 = pd.DataFrame({
    'Name': ['Charlie', 'David'],
    'Age': [22, 28]
})

concatenated_df = pd.concat([df1, df2], ignore_index=True)
print(concatenated_df)
```

Data Aggregation: Sum, Mean, Count, etc.

Data aggregation involves summarizing data using functions like sum, mean, count, and others to extract meaningful insights.

1. **Sum**: Adds up all the values in a series or DataFrame column.

```python
df['sales'].sum()  # Total of the 'sales' column
```

2. **Mean**: Calculates the average of the data.

```python
df['sales'].mean()  # Average sales value
```

3. **Count**: Counts the number of non-null entries.

```python
df['sales'].count()  # Total number of entries in 'sales'
```

4. **Other Aggregations**:

   ○ `max()`: Finds the maximum value.
   ○ `min()`: Finds the minimum value.

    - `median()`: Finds the median value.
    - `std()`: Calculates the standard deviation.
5. Example:

```python
df['sales'].agg(['sum', 'mean',
'count'])
```

Reading and Writing Data: CSV, Excel, JSON

Pandas makes it easy to handle different file formats for reading and writing data.

1. **CSV Files**:
    - Reading:

```python
df = pd.read_csv('data.csv')
```

    - Writing:

```python
df.to_csv('output.csv', index=False)
```

2. **Excel Files**:
    - Reading:

```python
df = pd.read_excel('data.xlsx')
```

- Writing:

```python
df.to_excel('output.xlsx', index=False)
```

3. **JSON Files**:

    - Reading:

```python
df = pd.read_json('data.json')
```

- Writing:

```python
df.to_json('output.json')
```

Handling Time Series Data

Time series data is indexed or organized by timestamps, making it suitable for trend analysis and forecasting.

1. **Converting to DateTime**: Convert a column to a `datetime` type for easier manipulation.

```python
df['date'] = pd.to_datetime(df['date'])
```

2. **Setting Date as Index**: Set the date column as the DataFrame index for time-based operations.

```python
df.set_index('date', inplace=True)
```

3. **Resampling**: Aggregate data over specific time intervals (e.g., monthly, yearly).

```python
df.resample('M').sum()  # Monthly
aggregation
```

4. **Date Filtering**: Filter rows based on specific date ranges.

```python
df.loc['2024-01-01':'2024-01-31']
```

5. **Time-based Features**: Extract components like year, month, or day from a datetime column.

Python

```
    df['year'] = df['date'].dt.year
    df['month'] = df['date'].dt.month
```