**PANDAS**

Pandas is a key library in Python for data manipulation and analysis. It’s particularly well-suited for working with structured data, like tables of data, where you need to organize, clean, and analyze the information**.**

**THINGS TO KNOW (basic):**

**Tuple:** A tuple in Python is an immutable, ordered collection of elements. Tuples are often used to group together related data of different types.

Denoted in parenthesis ( ) .

Eg: a = (1, 2, "apple", 3.14)

**List:** A list in Python is a versatile, mutable, and ordered collection of elements. Lists are one of the most commonly used data structures in Python, allowing you to store multiple items in a single variable. Lists can hold a mix of data types, including numbers, strings, and even other lists.

Denoted in Square Brackets [ ].

Eg: a= [1, 2, 3, "apple", 3.14, True]

**Dictionaries:** A dictionary in Python is an unordered, mutable collection of key-value pairs. Each key in a dictionary is unique and maps to a corresponding value, making dictionaries highly efficient for lookups, insertions, and deletions.

Have key and value.

Denoted in curly Braces {}

Eg: a = {"name": "Alice", "age": 25, "sstudent": True, "grades": [90, 88, 95]}

Here., name, age, students and grades are keys. Alice,25, True, [90,88,95] are value

**CONTENTS:**

1. **DATAFRAMES**
2. **ADD AND REMOVING COLUMNS**
3. **HANDLING MISSING DATA**
4. **AGGREGATION AND GROUPING**
5. **TIME SERIES**
6. **SORTING**
7. **FILTERING**

**DATAFRAMES:**

A Data Frame is essentially a table of data in a two-dimensional format, where rows represent records (or observations) and columns represent variables (or features). It is similar to a spreadsheet or SQL table.

**Creating a Data Frame:**

We can create a DataFrame from various sources like lists, dictionaries,csv files,etc…

**Syntax:**

For Dictionaries,lists:

df = pd.DataFrame(variable\_name)

For CSV Files:

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

**Basic operations in Data frames:**

**Accessing Data in a Data Frame:**

By Column Name:

1.For single column: df['column\_name']

2.For multiple column: df[['column\_name', 'column\_name']]

By Row Index:

1. Access the first row by index: df.loc[0]
2. Access the first row by position: f.iloc[0]

**Modifying a DataFrame:**

Adding a New Column:

df['Column\_name'] = [column\_values]

Renaming Columns:

df.rename(columns={'Name': 'Full Name'}, inplace=True)

**ADD AND REMOVING COLUMNS:**

Add a Column with a Constant Value:

df['new\_column'] = value

Add a Column with a Series:

df['new\_column'] = pd.Series(data)

Add a Column based on a Calculation:

df['new\_column'] = df['column1'] + df['column2']

Remove a Column using drop():

df = df.drop('column\_name', axis=1)

Remove Multiple Columns using drop():

df = df.drop(['column1', 'column2'], axis=1)

Remove a Column in Place using drop():

df.drop('column\_name', axis=1, inplace=True)

Remove Rows using drop():

df = df.drop(index\_name)

**HANDLING MISSING DATA:**

**Detecting Missing Data**

Check for Missing Values:

df.isnull()

Check for Non-Missing Values:

df.notnull()

Count Missing Values in Each ColumN:

df.isnull().sum()

**Handling Missing Data**

Drop Rows with Missing Values

df.dropna()

Drop Columns with Missing Values

df.dropna(axis=1)

Drop Rows with All Missing Values

df.dropna(how='all')

**Fill Missing Values with the Mean/Median/Mode**

df['column\_name'].fillna(df['column\_name'].mean(), inplace=True)

df['column\_name'].fillna(df['column\_name'].median(), inplace=True)

df['column\_name'].fillna(df['column\_name'].mode()[0], inplace=True)

**Forward Fill (Fill Missing Values with the Previous Value)**

df.fillna(method='ffill’)

**Backward Fill (Fill Missing Values with the Next Value)**

df.fillna(method='bfill')

**Replacing Specific Values with NaN**

import numpy as np

df.replace(specific\_value, np.nan, inplace=True)

**Filling Missing Values for Specific Columns**

df['column\_name'].fillna(value, inplace=True)

**AGGREGATION AND GROUPING**:

**Grouping Data:**

Basic Grouping:

df.groupby('column\_name')

Grouping by Multiple Columns:

df.groupby(['column1', 'column2'])

**Aggregation Functions:**

**Aggregation with a Single Function**

df.groupby('column\_name').agg('sum')

**Aggregation with Multiple Functions**

df.groupby('column\_name').agg(['sum', 'mean'])

**Applying Different Functions to Different Columns**

df.groupby('column\_name').agg({'column1': 'sum', 'column2': 'mean'})

**Common Aggregation Functions**

Sum of each group

df.groupby('column\_name').sum()

Mean of each group

df.groupby('column\_name').mean()

Count of each group

df.groupby('column\_name').count()

**Transforming Data with transform()**

df['new\_column'] =

df.groupby('column\_name')['column\_to\_transform'].transform('sum')

**Filtering Groups with filter()**

df.groupby('column\_name').filter(lambda x: x['column\_to\_check'].mean() > threshold)

**Applying Custom Functions**

df.groupby('column\_name').apply(custom\_function)

**GroupBy Object with Iteration**

for name, group in df.groupby('column\_name'):

print(name)

print(group)

**TIME SERIES:**

**Creating a time series:**

dates = pd.date\_range(date, periods=6, freq='D')

data = pd.Series([1, 3, 5, 7, 9, 11], index=dates)

print(data)

**Resampling:** Resampling involves changing the frequency of your time series data.

**Down sampling:**

a = data.resample('M').mean()

print(a) here., M = Monthly Data

**Up Sampling:**

b= data.resample('H').asfreq()

print(b) here., H = Hourly Data

**Shifting and Lagging:** Shift time series data forward or backward.

**Shift the series forward:**

shifted\_data = data.shift(1)

**Calculate the difference:**

diff\_data = data.diff(1)

here., 1 is the value to shift value and the difference

**Rolling Window:** Calculate rolling statistics like moving average.

Syntax:a = data.rolling(window= value).mean()

**Date Parsing:** Convert strings to datetime objects.

Syntax: a = ‘Date’

date\_obj = pd.to\_datetime(a)

**Time Series Indexing and Slicing:**

Select data for a specific date

Syntax: data\_on\_date = data['Date']

Slicing using date ranges

Syntax: data\_in\_range = data['start date':end date ']

**Time Zones:** Handling time zones in your time series data.

Localize the series to a specific time zone:

Syntax: localized\_data = data.tz\_localize(‘time zone name’)

Convert to another time zone:

Syntax: converted\_data = localized\_data.tz\_convert(‘time zone name’)

**Handling Missing Data:** Handling missing data points in your time series

Reindex to introduce missing dates:

Syntax:

new\_dates = pd.date\_range(‘date ', periods=8, freq='D') reindexed\_data = data.reindex(new\_dates)

Fill missing data

Syntax:

filled\_data = reindexed\_data.fillna(method='ffill')

print(filled\_data)

**SORTING:**

**Sorting by values:**

Syntax: Sorted\_data = data.sort\_values()

**Sorting by index:**

Syntax: Sorted\_by\_index = data.sort\_index()

**Sorting by values in column:**

Syntax: Sorted\_df = df.sort\_values(by='column\_name')

**Sorting by values in multiple columns:**

Syntax: sorted\_df = df.sort\_values(by = ['column1', 'column2'], ascending=[True, False])

**Sorting a DataFrame by Index:**

Sorting by the index:

Syntax: sorted\_df\_by\_index = df.sort\_index()

**Sorting in Descending Order:**

Sorting a Series in descending order:

Syntax: sorted\_data\_desc = data.sort\_values(ascending=False)

Sorting a DataFrame in descending order by column:

Syntax: sorted\_df\_desc = df.sort\_values(by='column\_name', ascending=False)

**Sorting with NaNs:**

Sorting, with NaNs placed last:

Syntax: sorted\_data\_with\_nan = data\_with\_nan.sort\_values()

Sorting, with NaNs placed first:

Syntax:

Sorted\_data\_with\_nan\_first = data\_with\_nan.sort\_values(na\_position='first')

**In-place Sorting**:

Sorting in place:

Syntax:

1. data.sort\_values(inplace=True)

1. df.sort\_values(by='A', inplace=True)

**FILTERING**

Filtering values greater than

Syntax: filtered\_data = data[data > value]

Filtering rows where column greater

Syntax: filtered\_df = df[df['column '] > value]

Filtering with multiple conditions

Syntax: filtered\_df\_and = df[(df['column1'] > value) & (df['column2'] < value)]

filtered\_df\_or = df[(df['colum1'] > value) | (df['column2'] < value)]

Filtering a DataFrame by a List of Values

Syntax: filtered\_df\_in = df[df['column'].isin([value])]

Filtering a DataFrame by String Matching

Filtering rows where Conditions:

Syntax: filtered\_df\_str = df\_str[df\_str['column'].str.[condition]]

Filtering with loc and iloc

Label-based filtering with loc

Syntax: filtered\_df\_loc = df.loc[df['column'] > value]

Position-based filtering with iloc

Syntax: filtered\_df\_iloc = df.iloc[start:end]

Filtering with Missing Values

Filtering rows where column is not nan

Syntax: filtered\_df\_notna = df\_with\_nan[df\_with\_nan['column'].notna()