UNIT III

**Getting Started with Pandas:** Introduction to Pandas Data Structures, Essential Functionality, Summarizing and Computing Descriptive Statistics.

**Data Loading, Storage and File Formats:** Reading and Writing Data in Text Format, Binary Data Formats, Interacting with Web APIs, Interacting with Databases.

**Getting Started with Pandas**

**1. Introduction to Pandas Data Structures**

Pandas is an essential library in Python for **data manipulation** and **analysis**. It provides fast, flexible, and expressive data structures for handling structured (tabular) data.

* **Series**: A **one-dimensional** labelled array similar to a list in Python but with the added feature of custom indexing.
* **DataFrame**: A **two-dimensional** tabular data structure, like an Excel sheet, with rows and columns. Each column in a DataFrame is a Series.

**Example: Series and DataFrame**

import pandas as pd

| # Creating a Series data = [10, 20, 30, 40] series = pd.Series(data, index=['a', 'b', 'c', 'd']) print("Series:\n", series)   # Creating a DataFrame data = {'Name': ['Alice', 'Bob', 'Charlie'], 'Age': [24, 27, 22]} df = pd.DataFrame(data) print("\nDataFrame:\n", df) |
| --- |

**Explanation:**

1. A **Series** acts like a dictionary with keys being the index labels. It is useful when you want to map values to specific labels.
2. A **DataFrame** allows working with rows and columns, making it ideal for analyzing data, running queries, and performing complex operations on datasets.

**2. Essential Functionality in Pandas**

Pandas provides several built-in methods for handling and manipulating data. Here are a few key operations you should know:

1. **Reindexing**: Changes or resets the index of a Series or DataFrame.
2. **Selection and Filtering**: Select data using **labels, indexes, or conditions**.
3. **Arithmetic Operations**: Perform operations across rows or columns, handling missing values gracefully.
4. **Handling Missing Data**: Use fillna() to replace missing values and dropna() to remove them.

**Example: Basic Operations**

| # Reindexing a Series new\_series = series.reindex(['a', 'b', 'c', 'd', 'e'], fill\_value=0) print("\nReindexed Series:\n", new\_series)   # Selecting rows based on conditions print("\nFiltering rows where Age > 23:\n", df[df['Age'] > 23])   # Filling missing values in DataFrame df['Age'] = df['Age'].fillna(df['Age'].mean()) |
| --- |

**Explanation:**

* **Reindexing** is useful when you want to align your data with a new index.
* **Selection and filtering** allow you to retrieve subsets of data that meet certain conditions, an essential operation in real-world data analysis.
* **Handling missing data** ensures that your computations are not affected by incomplete data.

**3. Summarizing and Computing Descriptive Statistics**

Descriptive statistics are essential to understanding the properties of your dataset. Pandas provides several statistical methods, including:

* count(): Number of non-null values.
* mean(): Average of the data.
* std(): Standard deviation of values.
* describe(): Summary of key statistics for numerical columns.

**Example: Descriptive Statistics**

| # Summary of statistics for the DataFrame print("\nDataFrame Statistics:\n", df.describe())   # Compute the mean of a specific column mean\_age = df['Age'].mean() print("\nMean Age:", mean\_age)   # Find the maximum value in the 'Age' column max\_age = df['Age'].max() print("\nMax Age:", max\_age) |
| --- |

**Explanation:**These statistical methods help you quickly understand patterns in your data, identify outliers, and prepare for more advanced analyses.

**Data Loading, Storage, and File Formats**

**1. Reading and Writing Data in Text Format**

The ability to load and store data in different formats is crucial in data science. Pandas supports formats like **CSV**, **JSON**, and **HTML**. Below is an example of working with **CSV (Comma-Separated Values)** files, one of the most common data formats.

**Example: CSV File Operations**

| # Writing to a CSV file df.to\_csv('data.csv', index=False)   # Reading from a CSV file df\_from\_csv = pd.read\_csv('data.csv') print("\nData from CSV:\n", df\_from\_csv) |
| --- |

**Explanation:**

* **CSV files** are widely used because they are easy to generate and work across many platforms.
* Pandas provides several parameters like sep, header, and index to customize how data is read and written.

**2. Binary Data Formats: Parquet and Excel**

Pandas also supports binary data formats, which are more efficient for large datasets. Parquet is a popular format for **big data processing**.

**Example: Parquet File Operations**

| # Writing DataFrame to Parquet df.to\_parquet('data.parquet')   # Reading from Parquet df\_from\_parquet = pd.read\_parquet('data.parquet') print("\nData from Parquet:\n", df\_from\_parquet) |
| --- |

**Explanation:**

* **Binary formats** like Parquet are compressed and allow faster reads and writes compared to text formats.
* They are commonly used in **big data frameworks** like Apache Spark and Hadoop.

**3. Interacting with Web APIs**

Using APIs, you can collect data from web services in **real-time** and convert it into DataFrames for analysis. The **requests** library is often used to make HTTP requests.

**Example: Fetching Data from an API**

| import requests   # API call to fetch JSON data url = 'https://jsonplaceholder.typicode.com/users' response = requests.get(url) data = response.json()   # Convert JSON to DataFrame df\_api = pd.DataFrame(data) print("\nData from API:\n", df\_api.head()) |
| --- |

**Explanation:**

* APIs allow dynamic data fetching, enabling applications to **stay updated** with the latest information.
* Pandas makes it easy to convert JSON or other structured data formats into DataFrames for further analysis.

**4. Interacting with Databases**

You can use Pandas to interact with relational databases like **SQLite**, **MySQL**, or **PostgreSQL**. This is useful for storing and retrieving large datasets.

**Example: Reading from a SQLite Database**

| import sqlite3   # Create a connection to a SQLite database conn = sqlite3.connect('example.db')   # Write DataFrame to SQL table df.to\_sql('users', conn, if\_exists='replace', index=False)   # Read data from SQL table df\_from\_sql = pd.read\_sql('SELECT \* FROM users', conn) print("\nData from SQL:\n", df\_from\_sql) |
| --- |

**Explanation:**

* Working with databases ensures efficient **storage and retrieval** of large datasets.
* Pandas' **SQL integration** allows you to directly use SQL queries within Python programs.

**Summary and Conclusion**

In this guide, we have covered:

1. **Pandas Data Structures**: Series and DataFrame, the two core components of Pandas.
2. **Essential Functionality**: Reindexing, selection, arithmetic operations, and handling missing data.
3. **Descriptive Statistics**: Quickly summarizing data using describe() and other methods.
4. **File Operations**: Reading from and writing to text and binary formats.
5. **Interacting with APIs and Databases**: Using web services and SQL databases for real-world data analysis.