Craft Software

Introduction to Pandas

Introduction to Pandas

What is Pandas?

Pandas is a powerful, open-source data analysis and manipulation library for Python. It provides flexible data structures for efficiently handling and analyzing large datasets.

Key Features

- Data Structures: Two primary data structures are provided by Pandas:
    - Series: A one-dimensional labeled array capable of holding any data type.
    - DataFrame: A two-dimensional labeled data structure with columns of potentially different types.
- Data Manipulation: Functions for data cleaning, transformation, and aggregation.
- Data Analysis: Tools for data exploration, aggregation, and visualization.
- Integration: Works well with other data science libraries like NumPy, SciPy, and Matplotlib.

Installation of pandas

To install Pandas, use the following pip command:

```
!pip install pandas

import pandas as pd

 #  import pandas in order to use pandas and use a short name of pd
```

1. Introduction to Pandas DataFrames

A DataFrame is a two-dimensional, labeled data structure in Pandas, similar to a spreadsheet or SQL table. It consists of rows and columns, where each column can have a different data type. DataFrames are one of the most commonly used structures for data manipulation in Pandas.

Examples:

- **Example 1: Creating a DataFrame from a Dictionary**

```
    import pandas as pd  # Import the pandas library

# Create a dictionary with data

data = {

  'Employee Name': ['Alice', 'Bob', 'Charlie'],  # List of employee names

  'Age': [28, 34, 25],  # List of ages

  'Department': ['HR', 'IT', 'Finance']  # List of departments

}

# Convert the dictionary into a DataFrame

df = pd.DataFrame(data)

# Display the DataFrame

print(df)
```

Explanation: We created a DataFrame by passing a dictionary where keys are column names, and values are lists of column data. Each key-value pair in the dictionary represents a column and its corresponding data.

- **Example 2: Creating a DataFrame from a List of Lists**

```
# Create a list of lists, where each inner list represents a row
data = [
    ['Alice', 28, 'HR'],
    ['Bob', 34, 'IT'],
    ['Charlie', 25, 'Finance']
]
# Convert the list of lists into a DataFrame and specify column names
df = pd.DataFrame(data, columns=['Employee Name', 'Age', 'Department'])

# Display the DataFrame
print(df)
```

Explanation: Each sublist in the list of lists represents a row in the DataFrame. We also specified column names to clearly identify each column of data.

- **Example 3: Creating a DataFrame from a List of Dictionaries**

```
# Create a list of dictionaries, where each dictionary represents a row
data = [
    {'Employee Name': 'Alice', 'Age': 28, 'Department': 'HR'},
    {'Employee Name': 'Bob', 'Age': 34, 'Department': 'IT'},
    {'Employee Name': 'Charlie', 'Age': 25, 'Department': 'Finance'}
]
# Convert the list of dictionaries into a DataFrame
df = pd.DataFrame(data)

# Display the DataFrame
print(df)
```

Explanation: Each dictionary represents a row in the DataFrame, with keys as column names and values as the row's data.

- **Example 4: Creating an Empty DataFrame and Adding Rows**

```
# Create an empty DataFrame with specified column names

df = pd.DataFrame(columns=['Employee Name', 'Age', 'Department'])

# Add rows one by one using the append method

df = df.append({'Employee Name': 'Alice', 'Age': 28, 'Department': 'HR'}, ignore_index=True)

df = df.append({'Employee Name': 'Bob', 'Age': 34, 'Department': 'IT'}, ignore_index=True)

df = df.append({'Employee Name': 'Charlie', 'Age': 25, 'Department': 'Finance'}, ignore_index=True)

# Display the DataFrame

print(df)
```

Explanation: We created an empty DataFrame with specified column names and added rows to it using the append method. The ignore_index=True parameter ensures that the index is reset for each new row.

2. Understanding Pandas Series

Description:

A Series in Pandas is a one-dimensional labeled array that can hold data of any type (integers, strings, floats, etc.). It's similar to a single column in a DataFrame. Series are useful for representing and manipulating a single column of data.

Examples:

- **Example 1: Creating a Series from a List**

```
# Create a Series from a list of employee names

employee_names = pd.Series(['Alice', 'Bob', 'Charlie'])

# Display the Series

print(employee_names)
```

This Series contains a list of employee names, with the default index starting from 0. A Series can be thought of as a single column of data.

- **Example 2: Accessing a Column from a DataFrame as a Series**

```
# Create a DataFrame
df = pd.DataFrame({
    'Employee Name': ['Alice', 'Bob', 'Charlie'],
    'Age': [28, 34, 25],
    'Department': ['HR', 'IT', 'Finance']
})
# Access the 'Age' column as a Series
ages = df['Age']
# Display the Series
print(ages)
```

Explanation: We extracted the 'Age' column from the DataFrame as a Series. This allows for operations and analysis on a single column of data.

- **Example 3: Creating a Series with Custom Index**

```
# Create a Series with custom index labels
departments = pd.Series(['HR', 'IT', 'Finance'], index=['Alice', 'Bob', 'Charlie'])
# Display the Series
print(departments)
```

Explanation: We created a Series with custom index labels, which makes it easier to access data based on meaningful labels rather than default numerical indices.

- **Example 4: Performing Operations on a Series**

```
# Create a Series of ages
ages = pd.Series([28, 34, 25])
# Add 1 to each age in the Series
ages_plus_one = ages + 1
# Display the modified Series
print(ages_plus_one)
```

Explanation: We performed a simple arithmetic operation (adding 1) on each element of the Series, demonstrating how Series support vectorized operations.

3. Reading Data with Pandas

Description:

Pandas can read data from a variety of formats such as CSV, Excel, JSON, and SQL databases. This functionality allows you to easily load and work with external datasets in a DataFrame.

Examples:

- **Example 1: Reading Data from a CSV File**

```
# Read data from a CSV file into a DataFrame
df = pd.read_csv('employees.csv')
# Display the first 5 rows of the DataFrame
print(df.head())
```

Explanation: We used read_csv to load data from a CSV file into a DataFrame. The head() method is used to display the first 5 rows of the DataFrame, giving a quick preview of the data.

- **Example 2: Reading Data from an Excel File**

```
# Read data from an Excel file into a DataFrame
df = pd.read_excel('employees.xlsx')

# Display the first 5 rows of the DataFrame
print(df.head())
```

Explanation: We used read_excel to load data from an Excel file into a DataFrame. This is useful when working with data stored in spreadsheets.

- **Example 3: Reading Data from a JSON File**

```
# Read data from a JSON file into a DataFrame
df = pd.read_json('employees.json')
# Display the first 5 rows of the DataFrame
print(df.head())
```

Explanation: JSON files are commonly used for data interchange, and we used read_json to load this data into a DataFrame for analysis.

- **Example 4: Reading Data from a SQL Database**

```
import sqlite3  # Import SQLite3 to connect to the database
# Establish a connection to the database
conn = sqlite3.connect('employees.db')
# Read data from a SQL table into a DataFrame
df = pd.read_sql_query('SELECT * FROM employees_table', conn)
# Display the first 5 rows of the DataFrame
print(df.head())
```

Explanation: Pandas can read data directly from SQL databases using the read_sql_query function, allowing for easy integration of database data into DataFrames.

4. Data Manipulation with Pandas

Description:

Data manipulation in Pandas involves modifying, adding, removing, or organizing data within a DataFrame. This is a crucial step in data analysis as it allows for the cleaning and preparation of data for further analysis.

Examples:

- **Example 1: Filtering Data Based on a Condition**

```
# Create a DataFrame
df = pd.DataFrame({
    'Employee Name': ['Alice', 'Bob', 'Charlie', 'David'],
    'Age': [28, 34, 25, 29],
    'Department': ['HR', 'IT', 'Finance', 'HR']
})
# Filter the DataFrame to include only employees in the HR department
hr_employees = df[df['Department'] == 'HR']
# Display the filtered DataFrame
print(hr_employees)
```

Explanation: We filtered the DataFrame to include only rows where the 'Department' column has a value of 'HR'. Filtering is commonly used to focus on specific subsets of data.

- **Example 2: Adding a New Column Based on a Calculation**

```
# Add a new column 'Years Until Retirement' calculated from the 'Age' column
df['Years Until Retirement'] = 65 - df['Age']
# Display the DataFrame with the new column
print(df)
```

Explanation: We added a new column 'Years Until Retirement' to the DataFrame by performing a calculation based on the 'Age' column. This shows how you can derive new information from existing data.

- **Example 3: Removing a Column**

```
# Remove the 'Department' column from the DataFrame
df = df.drop(columns=['Department'])
# Display the DataFrame after removing the column
print(df)
```

Explanation: We removed the 'Department' column using the drop method, which is useful when you want to clean up unnecessary data from your DataFrame.

- **Example 4: Renaming Columns**

```
# Rename the columns of the DataFrame
df = df.rename(columns={'Employee Name': 'Name', 'Age': 'Employee Age'})
# Display the DataFrame with renamed columns
print(df)
```

Explanation: We renamed the columns to more descriptive names using the rename method. Renaming columns can make the DataFrame more readable and understandable.

5. Basic Operations in Pandas

Pandas provides various basic operations for sorting, grouping, and summarizing data. These operations are essential for exploring and understanding the data before performing more complex analyses.

Examples:

- **Example 1: Sorting Data by a Single Column**

```
# Sort the DataFrame by the 'Age' column
sorted_df = df.sort_values(by='Age')
# Display the sorted DataFrame
print(sorted_df)
```

Explanation: We sorted the DataFrame by the 'Age' column in ascending order. Sorting is often used to organize data in a specific order for better analysis.

- **Example 2: Sorting Data by Multiple Columns**

```
# Sort the DataFrame by 'Department' and then by 'Age'
sorted_df = df.sort_values(by=['Department', 'Age'])
# Display the sorted DataFrame
print(sorted_df)
```

Explanation: We first sorted the DataFrame by 'Department', and within each department, we sorted by 'Age'. Sorting by multiple columns helps in organizing data hierarchically.

- **Example 3: Grouping Data and Calculating Aggregate Statistics**

```
# Group data by 'Department' and calculate the average age for each department

average_age_by_department = df.groupby('Department')['Age'].mean()

# Display the result

print(average_age_by_department)
```

Explanation: We grouped the data by 'Department' and calculated the average age for each group. Grouping is commonly used for summarizing and analyzing data across different categories.

- **Example 4: Counting Unique Values**

```
# Count the number of employees in each department

department_counts = df['Department'].value_counts()


# Display the result

print(department_counts)
```

Explanation: We used the value_counts method to count the number of occurrences of each unique value in the 'Department' column. This is useful for understanding the distribution of categorical data.