# Chapter.4.Pandas

Pandas is a powerful Python library that is widely used for data manipulation, analysis, and exploration. It provides flexible and efficient data structures—primarily DataFrame and Series—that allow you to work with structured data easily.

Pandas is built on top of NumPy and provides functionality for working with large datasets, handling missing data, filtering, grouping, and merging datasets, among other features. Let's explore Pandas with examples and detailed explanations of key operations.

## 1. Pandas Data Structures

Pandas has two primary data structures:

- **Series**: A one-dimensional labeled array capable of holding any data type (integers, strings, floats, etc.).
- **DataFrame**: A two-dimensional labeled data structure, similar to a table in a database or an Excel spreadsheet, with rows and columns.

### 1.1 Series

A Pandas Series is like a column in a spreadsheet or a one-dimensional array in NumPy. It consists of two main components: the data and the index.

**Example 1: Creating a Series**

```
import pandas as pd

# Creating a Series from a list
data = [10, 20, 30, 40, 50]
series = pd.Series(data, index=['a', 'b', 'c', 'd', 'e'])

print("Pandas Series:\n", series)
```

**Output:**

```
Pandas Series:
a    10
b    20
c    30
d    40
e    50
dtype: int64
```

**Explanation:**

- The Series is created from a list [10, 20, 30, 40, 50].
- The index is explicitly set to ['a', 'b', 'c', 'd', 'e'], allowing you to access elements using these labels.

### 1.2 DataFrame

A Pandas DataFrame is a 2D data structure with rows and columns, similar to a table. It can hold data of different types and allows for more complex operations like filtering, aggregation, and reshaping.

**Example 2: Creating a DataFrame**

```
# Creating a DataFrame from a dictionary
data = {
   'Name': ['Alice', 'Bob', 'Charlie', 'David'],
   'Age': [25, 30, 35, 40],
   'Salary': [50000, 60000, 70000, 80000]
}
df = pd.DataFrame(data)

print("Pandas DataFrame:\n", df)
```

**Output:**

```
Pandas DataFrame:
      Name  Age  Salary
0    Alice   25   50000
1      Bob   30   60000
2  Charlie   35   70000
3    David   40   80000
```

**Explanation:**

- The DataFrame is created from a dictionary where each key is a column and each value is a list of data for that column.
- The DataFrame has an automatic index (0, 1, 2, 3).

## 2. Basic Operations on DataFrames

You can perform a wide range of operations on DataFrames, such as accessing data, filtering, and modifying the structure.

### 2.1 Accessing Data

You can access individual columns, rows, or elements from a DataFrame.

**Example 3: Accessing Columns**

```
# Accessing a single column
name_column = df['Name']

# Accessing multiple columns
age_salary = df[['Age', 'Salary']]

print("Name Column:\n", name_column)
print("\nAge and Salary Columns:\n", age_salary)
```

**Output:**

```
Name Column:
0      Alice
1        Bob
2    Charlie
3      David
Name: Name, dtype: object

Age and Salary Columns:
   Age  Salary
0   25   50000
1   30   60000
2   35   70000
3   40   80000
```

**Explanation:**

- Accessing a single column returns a Series.
- Accessing multiple columns returns a new DataFrame.

### 2.2 Accessing Rows with loc[] and iloc[]

You can access rows based on index labels using loc[] or by integer positions using iloc[].

**Example 4: Accessing Rows**

```
# Accessing rows using loc (label-based)
row_bob = df.loc[1]

# Accessing rows using iloc (position-based)
first_two_rows = df.iloc[0:2]

print("Row for Bob (loc):\n", row_bob)
print("\nFirst two rows (iloc):\n", first_two_rows)
```

**Output:**

```
Row for Bob (loc):
Name       Bob
Age         30
Salary    60000
Name: 1, dtype: object

First two rows (iloc):
    Name  Age  Salary
0   Alice   25   50000
1     Bob   30   60000
```

**Explanation:**

- **loc[]** is used for label-based indexing, and it returns the row where the index label is 1.

- **iloc[]** is used for integer-based indexing and returns the first two rows of the DataFrame.

## 3. Modifying DataFrames

Pandas allows you to add new columns, modify existing ones, or drop rows/columns from the DataFrame.

### 3.1 Adding New Columns

You can easily add new columns to a DataFrame by assigning a new column name and values.

**Example 5: Adding a New Column**

```
# Adding a new column 'Bonus' to the DataFrame
df['Bonus'] = [5000, 6000, 7000, 8000]

print("DataFrame with New Column:\n", df)
```

**Output:**

```
DataFrame with New Column:
      Name  Age  Salary  Bonus
0    Alice   25   50000   5000
1      Bob   30   60000   6000
2  Charlie   35   70000   7000
3    David   40   80000   8000
```

**Explanation:**

- A new column Bonus is added to the DataFrame, and it holds the values [5000, 6000, 7000, 8000].

### 3.2 Dropping Rows or Columns

You can drop rows or columns from a DataFrame using the drop() method.

**Example 6: Dropping a Column**

```
# Dropping the 'Bonus' column
df_dropped = df.drop('Bonus', axis=1)
print("DataFrame after Dropping 'Bonus' Column:\n", df_dropped)
```

**Output:**

```
DataFrame after Dropping 'Bonus' Column:
      Name  Age  Salary
0    Alice   25   50000
1      Bob   30   60000
2  Charlie   35   70000
3    David   40   80000
```

**Explanation:**

- The drop() method is used to remove the Bonus column.
- The parameter axis=1 specifies that a column is being dropped. If you want to drop a row, use axis=0.

To drop a raw:

# Drop the last row (David)

df 2= df.drop(df.index[-1])

print(df)

or

df 3= df.iloc[:-1]

**If you want to drop multiple rows:**

df4 = df.drop([1, 3])

print(df4)

**Drop a row by condition**

Example: drop rows where Age > 30

df 5= df[df['Age'] <= 30]

print(df5)

Example: drop the row where Name == "Charlie"

df6 = df[df['Name'] != 'Charlie']

print(df6)

## 4. Handling Missing Data

Missing data is common in real-world datasets. Pandas provides several methods to handle missing values, such as filling them or dropping rows/columns with missing values.

### 4.1 Detecting Missing Data

You can detect missing data using the **isnull()** function, which returns a boolean DataFrame.

**Example 7: Detecting Missing Data**

# Introducing missing values
df.loc[1, 'Salary'] = None

# Checking for missing values

```
missing_data = df.isnull()
print("Missing Data:\n", missing_data)
```

**Output:**

```
Missing Data:
    Name   Age  Salary  Bonus
0  False  False   False  False
1  False  False    True  False
2  False  False   False  False
3  False  False   False  False
```

**Explanation:**

- A missing value is introduced at index 1 for the Salary column.
- **isnull()** returns a boolean DataFrame where True indicates a missing value.

### 4.2 Filling Missing Data

You can fill missing values using the **fillna()** method.

**Example 8: Filling Missing Data**

```
# Filling missing values in the 'Salary' column with the mean
df['Salary'] = df['Salary'].fillna(df['Salary'].mean())
print("DataFrame after Filling Missing Values:\n", df)
```

**Output:**

```
DataFrame after Filling Missing Values:
      Name  Age  Salary  Bonus
0    Alice   25  50000.0   5000
1      Bob   30  66666.67   6000
2  Charlie   35  70000.0   7000
3    David   40  80000.0   8000
```

**Explanation:**

- The missing value in the Salary column is filled with the mean salary using fillna().
- The mean salary is calculated as (50000 + 70000 + 80000) / 3 = 66666.67.

## 5. Filtering and Conditional Selection

You can filter rows in a DataFrame based on specific conditions, similar to SQL's WHERE clause.

### 5.1 Filtering Rows Based on Condition

You can filter rows based on column values using boolean indexing.

**Example 9: Filtering Rows**

```
# Filtering rows where Age > 30
```

```
filtered_df = df[df['Age'] > 30]

print("Rows where Age > 30:\n", filtered_df)
```

**Output:**

```
Rows where Age > 30:
      Name  Age   Salary  Bonus
2  Charlie   35  70000.0   7000
3    David   40  80000.0   8000
```

**Explanation:**

- The DataFrame is filtered to show only rows where the Age is greater than 30.

## 6. Grouping and Aggregation

Pandas allows you to group data by one or more columns and apply aggregate functions like **sum(), mean(), count(),** etc.

### 6.1 Grouping Data

You can group data by specific columns using the **groupby()** function and apply aggregate operations.

**Example 10: Grouping and Aggregation**

```
# Creating a new DataFrame
data = {
   'Department': ['HR', 'IT', 'HR', 'IT', 'Sales', 'Sales'],
   'Salary': [50000, 60000, 55000, 65000, 45000, 52000]
}
df_dept = pd.DataFrame(data)

# Grouping by 'Department' and calculating the mean salary
grouped = df_dept.groupby('Department').mean()

print("Grouped by Department:\n", grouped)
```

**Output:**

```
Grouped by Department:
              Salary
Department
HR          52500.0
IT          62500.0
Sales       48500.0
```

**Explanation:**

- The data is grouped by the Department column, and the mean salary for each department is calculated using mean().

## Conclusion

Pandas is an essential tool for data analysis and manipulation. Key features include:

- **Data structures**: Series (1D) and DataFrame (2D).
- **Basic operations**: Accessing and modifying data.
- **Handling missing data**: Detecting, filling, or dropping missing values.
- **Filtering**: Selecting rows based on conditions.
- **Grouping and aggregation**: Grouping data by one or more columns and applying aggregate functions.

Pandas makes it easy to handle and analyze large datasets efficiently, making it a fundamental library for data science and machine learning workflows.

# Exercises

## Basic Setup

```
import pandas as pd
import numpy as np
```

### Exercise 1: Create Data Structures

```
# Create a Series of temperatures for 5 days
# Create a DataFrame with student information (name, grade, subject)
```

***Solution:***

```
# Series
temperatures = pd.Series([72, 68, 75, 80, 78],
                         index=['Mon', 'Tue', 'Wed', 'Thu', 'Fri'])

# DataFrame
students = pd.DataFrame({
    'Name': ['John', 'Sarah', 'Mike', 'Emma'],
    'Grade': [85, 92, 78, 95],
    'Subject': ['Math', 'Science', 'Math', 'English']
})
print("Temperatures:")
print(temperatures)
print("\nStudents:")
print(students)
```

Outputs:

```
Temperatures:
Mon    72
Tue    68
Wed    75
Thu    80
Fri    78
dtype: int64

Students:
    Name  Grade    Subject
0   John     85       Math
1  Sarah     92    Science
2   Mike     78       Math
```

3 Emma 95 English

**Exercise2: Reading Data**

```
# Create sample data first
sample_data = {
   'Name': ['Alice', 'Bob', 'Charlie'],
   'Age': [25, 30, 35],
   'Salary': [50000, 60000, 70000]
}
df = pd.DataFrame(sample_data)

# Save to different formats
df.to_csv('sample.csv', index=False)
df.to_excel('sample.xlsx', index=False)

# Read from different formats
df_csv = pd.read_csv('sample.csv')
df_excel = pd.read_excel('sample.xlsx')

print("From CSV:")
print(df_csv)
print("\nFrom Excel:")
print(df_excel)
```

Outputs:

From CSV:

```
      Name  Age  Salary
0    Alice   25   50000
1      Bob   30   60000
2  Charlie   35   70000
```

From Excel:

```
      Name  Age  Salary
0    Alice   25   50000
1      Bob   30   60000
2  Charlie   35   70000
```