Manipulating Data with Pandas

Filtering Data

Filtering data is a fundamental operation in data analysis. Pandas allows you to filter data based on conditions using **Boolean indexing.**

Example Code:

```
import pandas as pd
data = {'Name': ['John', 'Alice', 'Bob'], 'Age': [25, 28, 22]}
df = pd.DataFrame(data)
# Drop rows with missing values
df.dropna()

# Fill missing values with a specific value
df.fillna(0)

# Filtering rows where Age is greater than 25
df[df['Age'] > 25]
```

Output

	Name	Age
1	Alice	28

The code is creating a DataFrame **df** from a dictionary **data**. The keys of the dictionary become column names, and the values become the data in those columns.

- The **dropna()** method is used to remove rows that contain missing (NaN) values.

  In this case, it's important to note that the **dropna()** method does not modify the original DataFrame in place. If you want to apply the changes to **df**, you need to either assign it back like **df = df.dropna()** or use the **inplace=True** parameter like **df.dropna(inplace=True)**.

- The fillna() method is applied to fill missing values in the DataFrame with 0.

  Similar to **dropna()**, if you want to apply the changes to **df**, you need to either assign it back (**df = df.fillna(0)**) or use **inplace=True** (**df.fillna(0, inplace=True)**).

- This code filters the DataFrame to include only those rows where the 'Age' column has a value greater than 25.

Let us now learn how data can be manipulated using Pandas.

Data Aggregation and Grouping

Data aggregation involves combining and summarizing data. Pandas supports grouping data based on specific criteria and applying aggregation functions.

Example Code:

```
import pandas as pd

# Extend the dataset
data = {'Name': ['John', 'Alice', 'Bob', 'Alice', 'John', 'Bob'],
        'Age': [25, 28, 22, 30, 26, 24],
        'Salary': [50000, 60000, 45000, 70000, 55000, 48000],
        'Department': ['HR', 'IT', 'IT', 'HR', 'HR', 'IT']}

df = pd.DataFrame(data)

# Original DataFrame
print("Original DataFrame:")
print(df)
```

Output

```
Original DataFrame:
    Name  Age  Salary Department
0   John   25   50000         HR
1  Alice   28   60000         IT
2    Bob   22   45000         IT
3  Alice   30   70000         HR
4   John   26   55000         HR
5    Bob   24   48000         IT
```

The code is creating a Python dictionary named data is created, representing columns ('Name', 'Age', 'Salary', 'Department') and their respective data. The pd.DataFrame() constructor is used to convert the Python dictionary data into a Pandas DataFrame named df. The output DataFrame represents a dataset with information about individuals, including their names, ages, salaries, and departments. The printed output provides a visual representation of the DataFrame.

We can do grouping in the three following ways:

1. Grouping by a single column and calculating the mean.
2. Grouping by multiple columns and calculating the sum.
3. Applying multiple aggregation functions.

Let's learn the ways of grouping data one by one.

1. **Grouping by a single column and calculating the mean:** In this method, data is grouped based on unique values in a single column, and the mean (average) is calculated for each group.

```
# Group by 'Department' and calculate the mean age for
each department
grouped_by_department =
df.groupby('Department')['Age'].mean()

print("\nMean Age by Department:")
print(grouped_by_department)
```

Output

```
Mean Age by Department:
Department
HR    27.000000
IT    24.666667
Name: Age, dtype: float64
```

The code uses the **groupby()** method to group the DataFrame df by the 'Department' column. It then selects the 'Age' column and calculates the mean age for each department using the mean() function. The result is a Pandas Series with department names as the index and mean ages as values. The print statements display the mean age for each department.

2. **Grouping by multiple columns and calculating the sum:** This approach involves grouping data based on combinations of values in multiple columns, and then calculating the sum for each resulting group.

```
# Group by 'Department' and 'Name' and calculate the sum
of salaries for each group
grouped_by_department_name = df.groupby(['Department',
'Name'])['Salary'].sum()

print("\nTotal Salary by Department and Name:")
print(grouped_by_department_name)
```

Output

```
Total Salary by Department and Name:
Department  Name
HR          Alice     70000
            John     105000
IT          Alice     60000
            Bob       93000
Name: Salary, dtype: int64
```

The code groups the DataFrame df by both 'Department' and 'Name' columns. It selects the 'Salary' column and calculates the sum of salaries for each group using the sum() function. The result is a Pandas Series with a **multi-level index** (Department and Name) and sum of salaries as values.The print statements display the total salary for each department and name combination.

3. **Applying multiple aggregation functions:** Data is grouped, and diverse aggregation functions (e.g., mean, sum, count) are applied concurrently, offering a comprehensive summary with various statistical measures.

```
# Group by 'Department' and apply multiple aggregation
functions
agg_functions = {'Age': 'mean', 'Salary': ['sum', 'mean']}

grouped_data = df.groupby('Department').agg(agg_functions)

print("\nAggregated Data by Department:")
print(grouped_data)
```

Output

```
Aggregated Data by Department:
                 Age  Salary
                mean     sum          mean
Department
HR         27.000000  175000  58333.333333
IT         24.666667  153000  51000.000000
```

The code groups the DataFrame df by the 'Department' column. It applies multiple aggregation functions to different columns using the agg() method. For 'Age', it calculates the mean. For 'Salary', it calculates both the sum and mean. The result is a new DataFrame with the department names as index and aggregated values. The print statements display the aggregated data for each department.

Let us now learn how we can handle missing data while preparing the data for analysis.

4. **Handling Missing Data:** Handling missing data is a crucial aspect of data analysis and manipulation.

   In Pandas, there are mainly two methods to deal with missing values:

   - dropping rows with missing values

- Filling missing values with specific values.

Let us consider a DataFrame with missing values:

Example Code:

```
import pandas as pd
import numpy as np

# Create a sample DataFrame with missing values
data = {'A': [1, 2, np.nan, 4],
        'B': [5, np.nan, np.nan, 8],
        'C': [9, 10, 11, 12]}

df = pd.DataFrame(data)

# Check for missing values
print("Original DataFrame:")
print(df)

# Fill missing values with the mean of each column
df_filled = df.fillna(df.mean())

print("\nDataFrame after filling missing values:")
print(df_filled)
```

Output

```
Original DataFrame:
     A    B   C
0  1.0  5.0   9
1  2.0  NaN  10
2  NaN  NaN  11
3  4.0  8.0  12

DataFrame after filling missing values:
          A    B   C
0  1.000000  5.0   9
1  2.000000  6.5  10
2  2.333333  6.5  11
3  4.000000  8.0  12
```

- A DataFrame **df** is created with three columns ('A', 'B', 'C') and four rows of data. Column 'A' has one missing value, and columns 'B' and 'C' have two missing values each.
- The **fillna()** method is used to fill missing values in the DataFrame **df** with the mean of each column. The result is stored in a new DataFrame named **df_filled**.

This code prints the DataFrame after filling missing values with the column means.