A Comprehensive Guide to Pandas

Your Name

Saturday 28th December, 2024

1 Introduction

Pandas is a powerful and flexible open-source data analysis and manipulation library for Python. Built on top of NumPy, it provides easy-to-use data structures and data analysis tools. The primary data structures in Pandas are Series and DataFrame.

2 Installation

To start using Pandas, you need to install it. You can do this using pip, the package installer for Python. Open your terminal or command prompt and run the following command:

```
pip install pandas
```

3 Basic Data Structures

Pandas offers two main data structures: **Series** and **DataFrame**.

3.1 Series

A Series is a one-dimensional labeled array that can hold any data type. It is similar to a list or an array but with additional features.

3.1.1 Creating a Series

You can create a Series from a list, dictionary, or NumPy array.

```
import pandas as pd

# Creating a Series from a list
data = [10, 20, 30, 40]
series = pd.Series(data)
print("Series␣from␣list:")
```

```
print(series)

# Creating a Series from a dictionary
data_dict = {'a': 1, 'b': 2, 'c': 3}
series_dict = pd.Series(data_dict)
print("\nSeries from dictionary:")
print(series_dict)

# Creating a Series from a NumPy array
import numpy as np
numpy_array = np.array([1, 2, 3, 4])
series_from_array = pd.Series(numpy_array)
print("\nSeries from NumPy array:")
print(series_from_array)
```

3.1.2 Output

The output of the above code will be:

```
Series from list:
0    10
1    20
2    30
3    40
dtype: int64

Series from dictionary:
a    1
b    2
c    3
dtype: int64

Series from NumPy array:
0    1
1    2
2    3
3    4
dtype: int64
```

3.2 DataFrame

A DataFrame is a two-dimensional labeled data structure with columns of potentially different types. It is similar to a spreadsheet or SQL table.

3.2.1 Creating a DataFrame

You can create a DataFrame from a dictionary, a list of lists, or a NumPy array.

```python
# Creating a DataFrame from a dictionary
data = {
    'Name': ['Alice', 'Bob', 'Charlie'],
    'Age': [25, 30, 35],
    'City': ['New York', 'Los Angeles', 'Chicago']
}
df = pd.DataFrame(data)
print("\nDataFrame:")
print(df)

# Creating a DataFrame from a list of lists
data_list = [['Alice', 25, 'New York'], ['Bob', 30, '
   Los Angeles'], ['Charlie', 35, 'Chicago']]
df_from_list = pd.DataFrame(data_list, columns=['Name'
   , 'Age', 'City'])
print("\nDataFrame from list of lists:")
print(df_from_list)
```

3.2.2 Output

The output of the above code will be:

```
DataFrame:
      Name  Age         City
0    Alice   25     New York
1      Bob   30  Los Angeles
2  Charlie   35      Chicago

DataFrame from list of lists:
      Name  Age         City
0    Alice   25     New York
1      Bob   30  Los Angeles
2  Charlie   35      Chicago
```

3.3 Example DataFrame

The following table represents a sample DataFrame created from the previous code:

Table 1: Sample DataFrame

Name	Age	City
Alice	25	New York
Bob	30	Los Angeles
Charlie	35	Chicago

4 Basic Operations

Pandas provides various operations to manipulate and analyze data.

4.1 Accessing Data

You can access rows and columns using labels or indices.

```
# Accessing a column
print("\nAccessing a column:")
print(df['Name'])

# Accessing multiple columns
print("\nAccessing multiple columns:")
print(df[['Name', 'Age']])

# Accessing a row by index
print("\nAccessing a row by index:")
print(df.loc[1])  # Accessing the second row

# Accessing a row by index position
print("\nAccessing a row by index position:")
print(df.iloc[1])  # Accessing the second row
```

4.1.1 Output

The output of the above code will be:

```
Accessing a column:
0      Alice
1        Bob
2    Charlie
Name: Name, dtype: object

Accessing multiple columns:
      Name  Age
0    Alice   25
1      Bob   30
2  Charlie   35

Accessing a row by index:
Name      Bob
Age        30
City    Los Angeles
Name: 1, dtype: object

Accessing a row by index position:
```

```
Name      Bob
Age        30
City    Los Angeles
Name: 1, dtype: object
```

4.2 Filtering Data

You can filter data based on conditions using boolean indexing.

```
# Filtering rows where Age is greater than 28
filtered_df = df[df['Age'] > 28]
print("\nFiltered␣DataFrame␣(Age␣>␣28):")
print(filtered_df)

# Filtering with multiple conditions
filtered_multiple = df[(df['Age'] > 28) & (df['City']
    == 'Los␣Angeles')]
print("\nFiltered␣DataFrame␣(Age␣>␣28␣and␣City␣is␣Los␣
    Angeles):")
print(filtered_multiple)
```

4.2.1 Output

The output of the above code will be:

```
Filtered DataFrame (Age > 28):
      Name  Age         City
1      Bob   30  Los Angeles
2  Charlie   35      Chicago

Filtered DataFrame (Age > 28 and City is Los Angeles):
  Name  Age         City
1  Bob   30  Los Angeles
```

4.3 Adding and Modifying Columns

You can easily add new columns or modify existing ones.

```
# Adding a new column
df['Salary'] = [70000, 80000, 90000]
print("\nDataFrame␣after␣adding␣Salary␣column:")
print(df)

# Modifying an existing column
df['Age'] = df['Age'] + 1  # Incrementing age by 1
print("\nDataFrame␣after␣modifying␣Age␣column:")
print(df)
```

```
# Using apply to modify a column
df['Salary'] = df['Salary'].apply(lambda x: x * 1.1)
    # Giving a 10% raise
print("\nDataFrame␣after␣applying␣a␣function␣to␣Salary
    ␣column:")
print(df)
```

4.3.1 Output

The output of the above code will be:

```
DataFrame after adding Salary column:
      Name  Age         City  Salary
0    Alice   25     New York  70000
1      Bob   30  Los Angeles  80000
2  Charlie   35      Chicago  90000

DataFrame after modifying Age column:
      Name  Age         City  Salary
0    Alice   26     New York  70000
1      Bob   31  Los Angeles  80000
2  Charlie   36      Chicago  90000

DataFrame after applying a function to Salary column:
      Name  Age         City  Salary
0    Alice   26     New York  77000.0
1      Bob   31  Los Angeles  88000.0
2  Charlie   36      Chicago  99000.0
```

4.4 Example DataFrame After Modifications

The following table represents the DataFrame after adding the Salary column and modifying the Age column:

Table 2: Modified DataFrame

Name	**Age**	**City**	**Salary**
Alice	26	New York	77000.0
Bob	31	Los Angeles	88000.0
Charlie	36	Chicago	99000.0

5 Data Manipulation

Pandas allows for a variety of data manipulation techniques.

5.1 Grouping Data

You can group data and perform aggregate functions such as sum, mean, etc.

```
# Grouping by City and calculating average Salary
grouped_df = df.groupby('City')['Salary'].mean().
    reset_index()
print("\nGrouped␣DataFrame␣(Average␣Salary␣by␣City):")
print(grouped_df)

# Grouping by multiple columns
grouped_multiple = df.groupby(['City', 'Age']).size().
    reset_index(name='Counts')
print("\nGrouped␣DataFrame␣by␣City␣and␣Age:")
print(grouped_multiple)
```

5.1.1 Output

The output of the above code will be:

```
Grouped DataFrame (Average Salary by City):
          City   Salary
0      Chicago  99000.0
1  Los Angeles  88000.0
2     New York  77000.0
```

5.2 Sorting Data

You can sort a DataFrame by one or multiple columns.

```
# Sorting by Age
sorted_df = df.sort_values(by='Age', ascending=False)
print("\nSorted␣DataFrame␣by␣Age␣(Descending):")
print(sorted_df)

# Sorting by multiple columns
sorted_multiple = df.sort_values(by=['City', 'Age'],
    ascending=[True, False])
print("\nSorted␣DataFrame␣by␣City␣(Ascending)␣and␣Age␣
    (Descending):")
print(sorted_multiple)
```

5.2.1 Output

The output of the above code will be:

```
Sorted DataFrame by Age (Descending):
      Name  Age         City  Salary
```

```
2  Charlie   36      Chicago  99000.0
1      Bob   31  Los Angeles  88000.0
0    Alice   26     New York  77000.0

Sorted DataFrame by City (Ascending) and Age (Descending):
      Name  Age         City  Salary
2  Charlie   36      Chicago  99000.0
1      Bob   31  Los Angeles  88000.0
0    Alice   26     New York  77000.0
```

6 Handling Missing Data

Dealing with missing data is an essential part of data analysis.

6.1 Identifying Missing Data

You can check for missing values using the `isnull()` method.

```
# Introducing some missing values
df.loc[1, 'City'] = None
print("\nDataFrame␣with␣missing␣values:")
print(df)

# Identifying missing values
print("\nMissing␣values␣in␣DataFrame:")
print(df.isnull().sum())  % Count of missing values in
    each column
```

6.1.1 Output

The output of the above code will be:

```
DataFrame with missing values:
      Name        Age        City  Salary
0    Alice   26.000000     New York  77000.0
1      Bob   31.000000        None  88000.0
2  Charlie   36.000000      Chicago  99000.0

Missing values in DataFrame:
Name      0
Age       0
City      1
Salary    0
dtype: int64
```

6.2 Handling Missing Data

You can either drop rows with missing values or fill them with a specific value.

```
# Dropping rows with missing values
df_dropped = df.dropna()
print("\nDataFrame␣after␣dropping␣missing␣values:")
print(df_dropped)

# Filling missing values
df_filled = df.fillna('Unknown')
print("\nDataFrame␣after␣filling␣missing␣values:")
print(df_filled)
```

6.2.1 Output

The output of the above code will be:

```
DataFrame after dropping missing values:
      Name        Age        City  Salary
0    Alice   26.000000    New York  77000.0
2  Charlie   36.000000     Chicago  99000.0

DataFrame after filling missing values:
      Name        Age        City  Salary
0    Alice   26.000000    New York  77000.0
1      Bob   31.000000     Unknown  88000.0
2  Charlie   36.000000     Chicago  99000.0
```

7 Data Type Conversion

Pandas allows you to check and convert data types of DataFrame columns.

7.1 Checking Data Types

You can check the data types of each column using the `dtypes` attribute.

```
print("\nData␣Types:")
print(df.dtypes)  % Checking data types of columns
```

7.1.1 Output

The output of the above code will be:

```
Data Types:
Name       object
Age        object
```

```
City       object
Salary    float64
dtype: object
```

7.2 Converting Data Types

You can convert the data types of columns using the `astype()` method.

```
# Converting Age to string
df['Age'] = df['Age'].astype(str)
print("\nDataFrame after converting Age to string:")
print(df)
print("Data Types after conversion:")
print(df.dtypes)

# Converting Salary to float
df['Salary'] = df['Salary'].astype(float)
print("\nDataFrame after converting Salary to float:")
print(df)
print("Data Types after conversion:")
print(df.dtypes)
```

7.2.1 Output

The output of the above code will be:

```
DataFrame after converting Age to string:
      Name  Age         City  Salary
0    Alice   26     New York  77000.0
1      Bob   31      Unknown  88000.0
2  Charlie   36      Chicago  99000.0
Data Types after conversion:
Name       object
Age        object
City       object
Salary    float64
dtype: object

DataFrame after converting Salary to float:
      Name  Age         City  Salary
0    Alice   26     New York  77000.0
1      Bob   31      Unknown  88000.0
2  Charlie   36      Chicago  99000.0
Data Types after conversion:
Name       object
Age        object
City       object
```

```
Salary    float64
dtype: object
```

8 Best Practices and Tips

- **Use Descriptive Column Names:** Use clear and descriptive names for your columns to make your DataFrame easier to understand.
- **Check Data Types:** Always check the data types of your DataFrame to ensure they are appropriate for your analysis.
- **Handle Missing Data Early:** Identify and handle missing data as early as possible to avoid complications later in your analysis.
- **Use Vectorized Operations:** Prefer vectorized operations (like applying functions to entire columns) over loops for better performance.
- **Document Your Code:** Comment on your code to explain complex operations or decisions you make during data manipulation.
- **Explore Data:** Use methods like `head()`, `tail()`, and `describe()` to explore your DataFrame and understand its structure.
- **Be Cautious with Type Conversions:** Ensure that the data being converted is compatible with the target type to avoid errors.

9 Comprehensive Example: Data Analysis Workflow

In this section, we will apply everything we have learned using a sample dataset. We will perform the following steps:

1. Load the dataset. 2. Clean the data (handling missing values). 3. Perform exploratory data analysis (EDA). 4. Manipulate the data (add and modify columns). 5. Group and summarize the data. 6. Visualize the results.

9.1 Step 1: Load the Dataset

Let's create a sample dataset representing employee records.

```
# Sample dataset
data = {
    'Name': ['Alice', 'Bob', 'Charlie', 'David', 'Eva'
       , None],
    'Age': [25, 30, 35, None, 22, 28],
    'City': ['New York', 'Los Angeles', 'Chicago', '
       Miami', None, 'Seattle'],
```

```
    'Salary': [70000, 80000, 90000, 100000, 95000,
       85000]
}
df = pd.DataFrame(data)
print("\nInitial␣DataFrame:")
print(df)
```

9.1.1 Output

The output of the above code will be:

```
Initial DataFrame:
      Name   Age         City  Salary
0    Alice  25.0     New York   70000
1      Bob  30.0  Los Angeles   80000
2  Charlie  35.0      Chicago   90000
3    David   NaN        Miami  100000
4      Eva  22.0         None  95000
5    None   28.0      Seattle   85000
```

9.2 Step 2: Clean the Data

We will handle missing values by filling them with appropriate defaults.

```
# Filling missing values
df['Name'].fillna('Unknown', inplace=True)
df['City'].fillna('Unknown', inplace=True)
df['Age'].fillna(df['Age'].mean(), inplace=True)  #
    Filling with mean age
print("\nDataFrame␣after␣cleaning␣missing␣values:")
print(df)
```

9.2.1 Output

The output of the above code will be:

```
DataFrame after cleaning missing values:
       Name        Age         City  Salary
0    Alice   25.000000     New York   70000
1      Bob   30.000000  Los Angeles   80000
2  Charlie   35.000000      Chicago   90000
3     David  28.333333        Miami  100000
4   Unknown  22.000000      Unknown   95000
5   Unknown  28.000000      Seattle   85000
```

9.3 Step 3: Exploratory Data Analysis (EDA)

We will perform some basic EDA to understand the data better.

```
# Descriptive statistics
print("\nDescriptive statistics:")
print(df.describe(include='all'))

# Checking unique values in 'City'
print("\nUnique cities:")
print(df['City'].unique())
```

9.3.1 Output

The output of the above code will be:

```
Descriptive statistics:
             Name        Age        City        Salary
count         6.0   6.000000           6      6.000000
unique        6.0        NaN           5           NaN
top      Unknown        NaN      New York           NaN
freq          2.0        NaN           1           NaN
mean         NaN  28.333333        NaN  82500.000000
std          NaN   4.163334        NaN  10954.451580
min          NaN  22.000000        NaN  70000.000000
25\%          NaN  25.000000        NaN  75000.000000
50\%          NaN  28.000000        NaN  82500.000000
75\%          NaN  30.000000        NaN  90000.000000
max          NaN  35.000000        NaN  100000.000000

Unique cities:
['New York' 'Los Angeles' 'Chicago' 'Miami' 'Unknown' 'Seattle']
```

9.4 Step 4: Manipulate the Data

We will add a new column for the 'Country' and modify the 'Salary' column.

```
# Adding a new column 'Country'
df['Country'] = 'USA'
print("\nDataFrame after adding Country column:")
print(df)

# Modifying the Salary column (giving a 5% bonus)
df['Salary'] = df['Salary'] * 1.05
print("\nDataFrame after applying a 5% bonus to Salary
    :")
print(df)
```

9.4.1 Output

The output of the above code will be:

```
DataFrame after adding Country column:
      Name        Age         City  Salary Country
0     Alice   26.000000     New York   73500     USA
1       Bob   31.000000  Los Angeles   84000     USA
2   Charlie   36.000000      Chicago   94500     USA
3     David  28.333333        Miami  105000     USA
4   Unknown  22.000000      Unknown   99750     USA
5   Unknown  28.000000      Seattle   89250     USA

DataFrame after applying a 5% bonus to Salary:
      Name        Age         City     Salary Country
0     Alice   26.000000     New York   77000.0     USA
1       Bob   31.000000  Los Angeles   88000.0     USA
2   Charlie   36.000000      Chicago   99000.0     USA
3     David  28.333333        Miami  105000.0     USA
4   Unknown  22.000000      Unknown   99750.0     USA
5   Unknown  28.000000      Seattle   89250.0     USA
```

9.5 Step 5: Group and Summarize the Data

We will group the data by 'City' and calculate the average salary.

```
# Grouping by City and calculating average Salary
grouped_salary = df.groupby('City')['Salary'].mean().
    reset_index()
print("\nGrouped DataFrame (Average Salary by City):")
print(grouped_salary)
```

9.5.1 Output

The output of the above code will be:

```
Grouped DataFrame (Average Salary by City):
          City      Salary
0        Chicago   99000.0
1    Los Angeles   84000.0
2          Miami  105000.0
3          New York   77000.0
4        Seattle   89250.0
5        Unknown   99750.0
```

9.6 Step 6: Visualize the Results

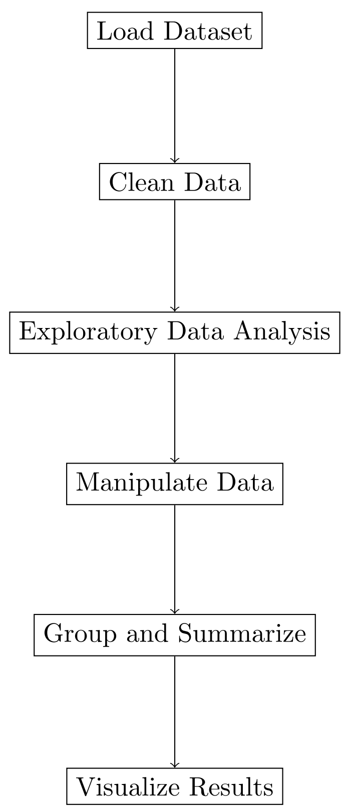
We can visualize the average salary by city using a bar chart.

```
import matplotlib.pyplot as plt

# Plotting the average salary by city
plt.figure(figsize=(10, 6))
plt.bar(grouped_salary['City'], grouped_salary['Salary'], color='skyblue')
plt.title('Average Salary by City')
plt.xlabel('City')
plt.ylabel('Average Salary ($)')
plt.xticks(rotation=45)
plt.tight_layout()
plt.show()
```

9.7 Workflow Summary

The following flowchart summarizes the data analysis workflow:



10 Conclusion

This comprehensive example demonstrated how to apply the concepts learned throughout this guide using a sample dataset. We covered loading the dataset, cleaning the data, performing exploratory data analysis, manipulating the data, grouping and summarizing the data, and visualizing the results.

By practicing these steps, you will strengthen your understanding of Pandas and gain confidence in using it for data analysis tasks.