Data Structures in Pandas

Series

In pandas, a series is a one-dimensional labeled array-like data structure. It can hold various data types and is often used to represent a column of data in a DataFrame

In [1]:
```
import pandas as pd

# Creating a Series from a list
data = [10, 20, 30, 40, 50]
series = pd.Series(data)
print(series)
```

```
0    10
1    20
2    30
3    40
4    50
dtype: int64
```

Series is ndarray-like

A Series in pandas is ndarray-like, which means you can perform array-like operations on it

In [2]:
```python
import pandas as pd

data = [10, 20, 30, 40, 50]
series = pd.Series(data)

# Array-like operations on the Series
print(series[1])        # Access element at index 1
print(series.mean())    # Calculate mean of elements
print(series.max())     # Find maximum value
print(series.sum())     # Sum all elements
```

```
20
30.0
50
150
```

Series is dict-like

In pandas, a Series is dict-like, meaning it's similar to a Python dictionary where values are accessed using keys.

In [3]:
```python
import pandas as pd

data = {'a': 10, 'b': 20, 'c': 30}
series = pd.Series(data)

# Dict-like operations on the Series
print(series['b'])      # Access value with key 'b'
print('c' in series)    # Check if key 'c' is present
print(series.keys())    # Get the keys
print(series.sum())  # Get the values
```

```
20
True
Index(['a', 'b', 'c'], dtype='object')
60
```

Vectorized Operations and Label Alignment with Series

Pandas Series support vectorized operations, which means you can perform operations on entire arrays without explicit looping.

In [4]:
```
import pandas as pd

# Creating Series
series1 = pd.Series([10, 20, 30], index=['a', 'b', 'c'])
series2 = pd.Series([5, 15, 25], index=['a', 'b', 'd'])

# Vectorized operations and label alignment
result_add = series1 + series2
result_mult = series1 * 2

print("Addition:")
print(result_add)

print("\nMultiplication:")
print(result_mult)
```

```
Addition:
a    15.0
b    35.0
c     NaN
d     NaN
dtype: float64

Multiplication:
a    20
b    40
c    60
dtype: int64
```

Name attribute

In pandas, the name attribute in a Series is used to assign a label or a name to the Series itself. This attribute can be helpful for identifying or providing context to the Series.

In [5]:
```python
import pandas as pd

data = [10, 20, 30, 40, 50]
series = pd.Series(data, name="MySeries")

print(series)
print("Series Name:", series.name)
```

```
0    10
1    20
2    30
3    40
4    50
Name: MySeries, dtype: int64
Series Name: MySeries
```

DataFrame

In pandas, a DataFrame is a two-dimensional labeled data structure that can hold heterogeneous data. It's similar to a table in a database or an Excel spreadsheet, with rows and columns.

In [6]:
```python
import pandas as pd

data = {'Name': ['Ali', 'Burhan', 'rana'],
        'Age': [25, 30, 22],
        'City': ['vehari', 'Bwp', 'Lahore']}

df = pd.DataFrame(data)

print(df)
```

```
     Name  Age    City
0     Ali   25  vehari
1  Burhan   30     Bwp
2    rana   22  Lahore
```

From dict of ndarrays / lists

you can create a DataFrame from a dictionary of ndarrays or lists. Each key becomes a column label, and the corresponding ndarray or list becomes the data for that column.

In [7]:
```
import pandas as pd

data = {'Name': ['Ali', 'Burhan', 'rana'],
        'Age': [25, 30, 22],
        'City': ['vehari', 'Bwp', 'Lahore']}

df = pd.DataFrame(data)

print(df)
```

```
     Name  Age    City
0     Ali   25  vehari
1  Burhan   30     Bwp
2    rana   22  Lahore
```

From structured or record array

In pandas, you can create a DataFrame from a structured or record array. These arrays are structured collections of data with named fields, which can be easily converted into DataFrame columns.

his approach is useful when dealing with structured data that comes from sources like databases or other data manipulation libraries.

In [8]:
```
import pandas as pd
import numpy as np

data = np.array([('Ali', 25, 'Bwp'),
                 ('Burhan', 30, 'Lahore'),
                 ('Umair', 22, 'Vehari')],
                dtype=[('Name', 'U10'), ('Age', int), ('City', 'U20')])

df = pd.DataFrame(data)

print(df)
```

```
     Name  Age    City
0     Ali   25     Bwp
1  Burhan   30  Lahore
2   Umair   22  Vehari
```

From a list of dicts

In pandas, you can create a DataFrame from a list of dictionaries. Each dictionary represents a row, and the keys become the column labels.

This method is handy when you have data in a structured dictionary format and want to convert it into tabular form.

In [9]:
```
import pandas as pd

data = [{'Name': 'Ali', 'Age': 25, 'City': 'Bwp'},
        {'Name': 'Burhan', 'Age': 30, 'City': 'Lahore'},
        {'Name': 'Uzair', 'Age': 22, 'City': 'Vehari'}]

df = pd.DataFrame(data)

print(df)
```

```
     Name  Age    City
0     Ali   25     Bwp
1  Burhan   30  Lahore
2   Uzair   22  Vehari
```

From a dict of tuples

In pandas, you can create a DataFrame from a dictionary of tuples. Each tuple represents a row, and the keys of the dictionary become the column labels.

This method can be useful when you have data in a tuple-based structure and want to convert it into a DataFrame.

In [10]:
```
import pandas as pd

data = {'Row1': ('Ali', 25, 'Multan'),
        'Row2': ('Umair', 30, 'Karachi'),
        'Row3': ('Ateeq', 22, 'Lahore')}


df = pd.DataFrame(data.values(), columns=['Name', 'age', 'city'], index = data.keys())
print(df)
```

```
       Name  age     city
Row1    Ali   25   Multan
Row2  Umair   30  Karachi
Row3  Ateeq   22   Lahore
```

From a Series

You can create a DataFrame from a Series in pandas. Each Series will become a column in the DataFrame

This method is useful when you have Series data and want to organize it into a DataFrame.

In [11]:
```
import pandas as pd

data = {'Name': pd.Series(['Ali', 'Burhan', 'Saeed']),
        'Age': pd.Series([25, 30, 22]),
        'City': pd.Series(['New York', 'London', 'Los Angeles'])}

df = pd.DataFrame(data)

print(df)
```

```
     Name  Age         City
0     Ali   25     New York
1  Burhan   30       London
2   Saeed   22  Los Angeles
```

From a list of namedtuples

In pandas, you can create a DataFrame from a list of namedtuples. Each namedtuple represents a row, and the fields of the namedtuple become the column labels.

This method is useful when you have data in namedtuple format and want to convert it into a DataFrame.

In [12]:
```python
import pandas as pd
from collections import namedtuple

# Define a namedtuple

Person = namedtuple('Person', ['Name', 'Age', 'City'])

# Create a list of namedtuples
data = [Person('Ali', 25, 'New York'),
        Person('Burhan', 30, 'London'),
        Person('Ahmad', 22, 'Los Angeles')]

df = pd.DataFrame(data)

print(df)
```

```
     Name  Age         City
0     Ali   25     New York
1  Burhan   30       London
2   Ahmad   22  Los Angeles
```

From a list of dataclasses

In pandas, you can create a DataFrame from a list of dataclasses. Each dataclass instance represents a row, and the fields of the dataclass become the column labels.

This method is useful when you have data in dataclass format and want to convert it into a DataFrame.

In [13]:
```python
import pandas as pd
from dataclasses import dataclass
from dataclasses import dataclass

@dataclass
class Person:
    Nmae: str
    Age: int
    City: str

# Create a list of dataclass instances
data = [Person('Ali', 25, 'Melbroune'),
        Person('Bilal', 30, 'Karachi'),
        Person('Waseem', 22, 'Lahore')]


df = pd.DataFrame([vars(person) for person in data])
print(df)
```

```
     Nmae  Age       City
0     Ali   25  Melbroune
1   Bilal   30    Karachi
2  Waseem   22     Lahore
```

Alternate constructors

In pandas, alternate constructors are methods that provide different ways to create a DataFrame. These methods offer flexibility in how you can organize and input your data.

These alternate constructors cater to different scenarios and data formats, making it easier to create DataFrames from diverse sources and structures.

1- pd.DataFrame.from_dict()*

Creates a DataFrame from a dictionary where keys become columns and values form data in columns.

In [14]:
```python
import pandas as pd

data = {'Name': ['Ali', 'Bilal', 'Rana'],
        'Age': [25, 30, 22]}

df = pd.DataFrame.from_dict(data)
print(df)
```

```
    Name  Age
0    Ali   25
1  Bilal   30
2   Rana   22
```

2- pd.DataFrame.from_records()

Creates a DataFrame from a list of records (tuples, dictionaries, or namedtuples).

In [15]:
```python
import pandas as pd

records = [('Ali', 25), ('Bilal', 30), ('Ahmad', 22)]

df = pd.DataFrame.from_records(records, columns=['Name', 'Age'])
print(df)
```

```
    Name  Age
0    Ali   25
1  Bilal   30
2  Ahmad   22
```

3- pd.DataFrame.from_excel()

Creates a DataFrame from an Excel file.

4- pd.DataFrame.from_records()

Creates a DataFrame from a list of records (e.g., namedtuples or dictionaries) and allows specifying column names.

In [16]:
```python
import pandas as pd
from collections import namedtuple

Person = namedtuple('Person', ['Name', 'Age'])
records = [Person('Ali', 25), Person('Bilal', 30), Person('siam', 22)]

df = pd.DataFrame.from_records(records, columns= Person._fields)
print(df)
```

```
    Name  Age
0    Ali   25
1  Bilal   30
2   siam   22
```

Column selection, addition, deletion

1- Column Selection

To select one or more columns from a DataFrame, you can use the column names within square brackets or by using the dot notation.

In [17]:
```python
import pandas as pd

data = {'Name': ['Ali', 'Uzair', 'Alina'],
        'Age': [25, 30, 22]}
df = pd.DataFrame(data)

# Select single column
age_column = df['Age']
print(age_column)

# Select multiple columns
name_and_age = df[['Name', 'Age']]
print(name_and_age)
```

```
0    25
1    30
2    22
Name: Age, dtype: int64
    Name  Age
0    Ali   25
1  Uzair   30
2  Alina   22
```

2- Column Addition

To add a new column to a DataFrame, you can directly assign values to a new column name.

In [18]:
```python
import pandas as pd

data = {'Name': ['Ali', 'Bilal', 'Waseem'],
        'Age': [25, 30, 22]}
df = pd.DataFrame(data)

# Add a new column
df['City'] = ['New York', 'London', 'Los Angeles']
print(df)
```

```
     Name  Age         City
0     Ali   25     New York
1   Bilal   30       London
2  Waseem   22  Los Angeles
```

3- Column Deletion

To delete a column from a DataFrame, you can use the drop() method with the appropriate column name and the axis parameter set to 1.

In [19]:
```python
import pandas as pd

data = {'Name': ['Ali', 'Bilal', 'Alina'],
        'Age': [25, 30, 22],
        'City': ['New York', 'London', 'Los Angeles']}
df = pd.DataFrame(data)

# Delete a column
df = df.drop('City', axis=1)
print(df)
```

```
    Name  Age
0    Ali   25
1  Bilal   30
2  Alina   22
```

Assigning new columns in method chains

In pandas, you can efficiently assign new columns to a DataFrame using method chaining. This allows you to perform multiple

In [20]:
```
import pandas as pd

data = {'Name': ['Ali', 'Bilal', 'Siam'],
        'Age': [25, 30, 22]}
df = pd.DataFrame(data)

# Method chaining to assign new columns

df = df.assign(
    City=['Lahore', 'Bwp', 'Fsd'],
    Status=['Active', 'Inactive', 'Active']
)

print(df)
```

```
    Name  Age    City    Status
0    Ali   25  Lahore    Active
1  Bilal   30     Bwp  Inactive
2   Siam   22     Fsd    Active
```

Indexing / selection

In pandas, indexing and selection refer to the process of accessing specific rows and columns in a DataFrame.

1- Indexing Rows and Columns by Label

You can use labels to index and select rows and columns using .loc[].

In [21]:
```python
import pandas as pd

data = {'Name': ['Ali', 'Bilal', 'Ahmad'],
        'Age': [25, 30, 22]}
df = pd.DataFrame(data, index=['A', 'B', 'C'])

# Select rows and columns using labels
selected_data = df.loc['A', 'Age']
print(df)
print(selected_data)
```

```
    Name  Age
A    Ali   25
B  Bilal   30
C  Ahmad   22
25
```

2- Indexing Rows and Columns by Position

You can use integer positions to index and select rows and columns using .iloc[].

In [22]:
```python
import pandas as pd

data = {'Name': ['Ali', 'Burhan', 'Charlie'],
        'Age': [25, 30, 22]}
df = pd.DataFrame(data)

# Select rows and columns using positions
selected_data = df.iloc[0, 1]
print(selected_data)
```

```
25
```

3- Conditional Selection

You can use conditional expressions to select rows that meet specific criteria.

In [23]:
```python
import pandas as pd

data = {'Name': ['Ali', 'Bilal', 'Ali'],
        'Age': [25, 30, 22]}
df = pd.DataFrame(data)

# Conditional selection
selected_rows = df[df['Age'] > 24]
print(selected_rows)
```

```
    Name  Age
0    Ali   25
1  Bilal   30
```

4- Selecting Columns

You can select one or more columns using column names.

In [24]:
```python
import pandas as pd

data = {'Name': ['Ali', 'Bilal', 'Alina'],
        'Age': [25, 30, 22]}
df = pd.DataFrame(data)

# Select specific columns
selected_columns = df[['Name', 'Age']]
print(selected_columns)
```

```
    Name  Age
0    Ali   25
1  Bilal   30
2  Alina   22
```

5- Slicing Rows

You can slice rows using integer positions

In [25]:
```python
import pandas as pd

data = {'Name': ['Ali', 'Burhan', 'Bilawal'],
        'Age': [25, 30, 22]}
df = pd.DataFrame(data)

# Slice rows using integer positions
sliced_rows = df[1:3]
print(sliced_rows)
```

```
      Name  Age
1   Burhan   30
2  Bilawal   22
```

Data alignment and arithmetic

In pandas, data alignment and arithmetic refer to how operations between Series and DataFrames are handled automatically, aligning data based on labels. This ensures that calculations are performed correctly even when the indices are not identical.

1- Data Alignment

When performing operations between pandas objects (Series or DataFrames), data is aligned based on their indices. Missing values are introduced where indices don't match, ensuring consistent alignment.

In [26]:
```python
import pandas as pd

data1 = {'Name': ['Ali', 'Bilal', 'Ahmad'],
         'Age': [25, 30, 22]}
df1 = pd.DataFrame(data1, index=['A', 'B', 'C'])

data2 = {'Age': [26, 28, 23],
         'City': ['New York', 'London', 'Los Angeles']}
df2 = pd.DataFrame(data2, index=['B', 'C', 'D'])

# Add two DataFrames with different indices

result = df1 + df2
print(result)
```

```
    Age  City  Name
A   NaN   NaN   NaN
B  56.0   NaN   NaN
C  50.0   NaN   NaN
D   NaN   NaN   NaN
```

2- Arithmetic Operations

Arithmetic operations between Series or DataFrames are handled element-wise, with alignment based on indices. This includes addition, subtraction, multiplication, and division.

In [27]:
```python
import pandas as pd

series1 = pd.Series([10, 20, 30], index=['a', 'b', 'c'])
series2 = pd.Series([5, 15, 25], index=['b', 'c', 'd'])

# Perform arithmetic operations with Series
addition = series1 + series2
multiplication = series1 * 2
division = series2 / series1

print("Addition:")
print(addition)

print("\nMultiplication:")
print(multiplication)

print("\nDivision:")
print(division)
```

```
Addition:
a     NaN
b    25.0
c    45.0
d     NaN
dtype: float64

Multiplication:
a    20
b    40
c    60
dtype: int64

Division:
a     NaN
b    0.25
c    0.50
d     NaN
dtype: float64
```

Transposing

In pandas, transposing a DataFrame means swapping rows and columns, effectively flipping the DataFrame along its diagonal. This can be useful for reshaping data or changing the orientation for better visualization or analysis.

In [28]:
```
import pandas as pd

data = {'Name': ['Ali', 'Bilal', 'Siam'],
        'Age': [25, 30, 22]}
df = pd.DataFrame(data)

# Transpose the DataFrame
transposed_df = df.T

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

print("\nTransposed DataFrame:")
print(transposed_df)
```

```
Original DataFrame:
    Name  Age
0    Ali   25
1  Bilal   30
2   Siam   22

Transposed DataFrame:
        0      1     2
Name  Ali  Bilal  Siam
Age    25     30    22
```

DataFrame interoperability with NumPy functions*

Pandas DataFrames seamlessly integrate with NumPy, allowing you to use NumPy functions for various operations on DataFrame data. This interoperability simplifies data manipulation and analysis

1- Applying NumPy Functions on DataFrame Columns

In [29]:
```python
import pandas as pd
import numpy as np

data = {'A': [1, 2, 3],
        'B': [4, 5, 6]}
df = pd.DataFrame(data)

# Using NumPy functions on DataFrame columns
result = np.sqrt(df['A']) + np.exp(df['B'])
print(result)
```

```
0     55.598150
1    149.827373
2    405.160844
dtype: float64
```

2- Creating NumPy Arrays from DataFrame Columns

In [30]:
```python
import pandas as pd
import numpy as np

data = {'A': [1, 2, 3],
        'B': [4, 5, 6]}
df = pd.DataFrame(data)

# Creating NumPy arrays from DataFrame columns
array_A = np.array(df['A'])
array_B = np.array(df['B'])

print(array_A)
print(array_B)
```

```
[1 2 3]
[4 5 6]
```

3- Using NumPy Functions for Aggregation

In [31]:
```python
import pandas as pd
import numpy as np

data = {'A': [1, 2, 3],
        'B': [4, 5, 6]}
df = pd.DataFrame(data)

# Using NumPy functions for aggregation
mean_A = np.mean(df['A'])
sum_B = np.sum(df['B'])

print("Mean of A:", mean_A)
print("Sum of B:", sum_B)
```

```
Mean of A: 2.0
Sum of B: 15
```

4- Using NumPy Broadcasting with DataFrame Columns

In [32]:
```python
import pandas as pd
import numpy as np

data = {'A': [1, 2, 3],
        'B': [4, 5, 6]}
df = pd.DataFrame(data)

# Using NumPy broadcasting with DataFrame columns
result = df['A'] + np.array([10, 20, 30])
print(result)
```

```
0    11
1    22
2    33
Name: A, dtype: int64
```

In [ ]: