DE235 Soumyadeep Sinha
WEEK 2 - DAY 3 - 08-11-2024

PYTHON : Numpy, Pandas

NumPy

NumPy is a powerful library in Python for numerical computations, particularly efficient for large arrays and matrices. Its core functionality is built around the **array** object, which enables high-performance operations on large data sets. It provides a suite of mathematical functions to operate on these arrays, offering a solid foundation for scientific computing and data analysis.

1. NumPy Arrays

Unlike Python lists, NumPy arrays are more memory efficient and allow complex operations with ease.

Example

```
import numpy as np

# Creating an array
arr = np.array([1, 2, 3, 4, 5])
print(arr)  # Output: [1 2 3 4 5]
```

Key Features of Arrays:

- **ndarray**: Homogeneous n-dimensional array object.
- **Shape**: Returns the dimensions of an array.
- **Dtype**: Specifies the type of elements in an array.

```
print(arr.shape)  # Output: (5,)
print(arr.dtype)  # Output: int32 (may vary by system)
```

2. Array Creation Methods

NumPy offers various functions to create arrays:

Examples

- Zeros and Ones:

```
zeros_array = np.zeros((3, 3))
ones_array = np.ones((2, 4))
```

- Arange and Linspace:

```
arange_array = np.arange(0, 10, 2)  # Output: [0, 2, 4, 6, 8]
linspace_array = np.linspace(0, 5, 10)  # 10 evenly spaced values
```

- Random:

```
random_array = np.random.random((3, 3)
```

3. Array Indexing and Slicing

Indexing and slicing work similarly to Python lists but support multi-dimensional arrays.

Example

```
matrix = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
print(matrix[1, 2])  # Accessing element at row 1, column 2 -> Output: 6
print(matrix[:, 1])  # Accessing all elements in column 1 -> Output: [2, 5, 8]
```

4. Array Manipulation

NumPy allows reshaping, stacking, and other modifications:

Examples

Reshape:

```
reshaped_matrix = np.arange(1, 10).reshape(3, 3)
```

Concatenate:

```
arr1 = np.array([1, 2, 3])
arr2 = np.array([4, 5, 6])
concatenated = np.concatenate((arr1, arr2))  # Output: [1, 2, 3, 4, 5, 6]
```

Stacking:

```
stacked = np.vstack((arr1, arr2))
```

5. Mathematical Operations

NumPy provides element-wise operations and linear algebra functions.

Examples

Element-wise Operations:

```
arr = np.array([1, 2, 3])
result = arr * 2  # Output: [2, 4, 6]
```

Aggregate Functions:

```
array = np.array([1, 2, 3, 4, 5])
print(np.sum(array))     # Sum -> Output: 15
print(np.mean(array))    # Mean -> Output: 3.0
print(np.min(array))     # Minimum -> Output: 1
print(np.max(array))     # Maximum -> Output: 5
```

Dot Product:

```
arr1 = np.array([1, 2])
arr2 = np.array([3, 4])
dot_product = np.dot(arr1, arr2)  # Output: 11 (1*3 + 2*4)
```

6. Broadcasting

Broadcasting allows operations on arrays of different shapes.

Example

```
arr1 = np.array([[1, 2, 3], [4, 5, 6]])
arr2 = np.array([1, 2, 3])
result = arr1 + arr2  # Broadcasting: Adds each row of arr1 with arr2
```

7. Boolean Indexing and Filtering

Boolean indexing allows filtering elements based on conditions.

Example

```
arr = np.array([1, 2, 3, 4, 5])
filtered = arr[arr > 2]  # Output: [3, 4, 5]
```

8. Advanced Array Manipulations

NumPy provides advanced methods like `np.where`, `np.unique`, and more.

Examples

np.where:

```
arr = np.array([1, 2, 3, 4, 5])
result = np.where(arr > 3, arr, -1)  # Replace elements not greater than 3 with -1
```

Np.unique

```
arr = np.array([1, 2, 2, 3, 3, 3])
unique_values = np.unique(arr)  # Output: [1, 2, 3]
```

9. Linear Algebra

NumPy includes a comprehensive suite for linear algebra computations.

Examples

Matrix Multiplication:

```
mat1 = np.array([[1, 2], [3, 4]])
mat2 = np.array([[5, 6], [7, 8]])
product = np.dot(mat1, mat2)
```

Inverse and Determinant:

```
matrix = np.array([[1, 2], [3, 4]])
inv_matrix = np.linalg.inv(matrix)
determinant = np.linalg.det(matrix)
```

10. Statistics and Probability

NumPy offers functions for statistical calculations.

Example

```
data = np.array([1, 2, 3, 4, 5, 6])
print(np.mean(data))      # Mean
print(np.std(data))       # Standard deviation
print(np.median(data))    # Median
print(np.percentile(data, 25))  # 25th percentile
```

Summary of Important Methods

1. **Creation: `np.array`, `np.zeros`, `np.ones`, `np.arange`, `np.linspace`**
2. **Shape Manipulation: `reshape`, `ravel`, `transpose`**
3. **Concatenate and Stack: `concatenate`, `vstack`, `hstack`**
4. **Mathematics: `sum`, `mean`, `std`, `dot`, `sqrt`, `exp`**
5. **Conditionals: `where`, `unique`, `nonzero`, `logical_and`, `logical_or`**
6. **Algebra: `inv`, `det`, `eig`, `dot`, `cross`**
7. **Statistics: `mean`, `std`, `percentile`, `median`, `var`**

Pandas

Pandas is a Python library designed for **data manipulation** and analysis. It provides DataFrame and Series objects, which allow for efficient handling and transformation of data. It's highly optimized for working with structured data and is commonly used in data analysis, data wrangling, and data science.

1. Pandas Series

A Series is a one-dimensional array-like object with labeled axes.

Example

```
import pandas as pd

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

Output:

```
0    10
1    20
```

```
2    30
3    40
dtype: int64
```

2. Pandas DataFrame

A DataFrame is a two-dimensional, size-mutable, and potentially heterogeneous data structure with labeled axes (rows and columns).

Example

```
data = {
    'Name': ['Alice', 'Bob', 'Charlie'],
    'Age': [25, 30, 35],
    'City': ['New York', 'Chicago', 'Los Angeles']
}
df = pd.DataFrame(data)
print(df)
```

Output:

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

3. Reading and Writing Data

Pandas can handle multiple file formats including CSV, Excel, and SQL databases.

Examples

Reading CSV:

```
df = pd.read_csv('data.csv')
```

Writing to CSV:

```
df.to_csv('output.csv', index=False)
```

4. DataFrame Basic Operations

Some useful operations for inspecting data include:

Examples

Head and Tail:

```
print(df.head(2))  # Shows first two rows
print(df.tail(2))  # Shows last two rows
```

Info and Describe:

```
print(df.info())      # Summary of DataFrame
print(df.describe())  # Statistical summary of numerical columns
```

Shape and Columns:

```
print(df.shape)    # Output: (3, 3) for 3 rows and 3 columns
print(df.columns)  # Output: Index(['Name', 'Age', 'City'], dtype='object')
```

5. Data Selection and Filtering

Data selection can be done by column indexing, boolean indexing, and by using conditions.

Examples

Selecting Columns:

```
print(df['Name'])           # Select single column
print(df[['Name', 'City']]) # Select multiple columns
```

Filtering Rows:

```
print(df[df['Age'] > 25])  # Filter rows where Age > 25
```

Using loc and iloc:

```
print(df.loc[0])       # Select by row label
print(df.iloc[1, 2])    # Select by row/column index
```

6. Data Cleaning

Pandas provides functions to handle missing values, duplicates, and data transformation.

Examples

Handling Missing Values:

```
df['Age'].fillna(df['Age'].mean(), inplace=True)  # Fill NaNs with column mean
df.dropna(inplace=True)                           # Drop rows with NaNs
```

Removing Duplicates:

```
df.drop_duplicates(inplace=True)
```

Renaming Columns:

```
df.rename(columns={'Name': 'Full Name'}, inplace=True)
```

7. Data Transformation

Pandas supports various ways to transform data, such as applying functions and mapping.

Examples

```
df['Age'] = df['Age'].apply(lambda x: x + 1)  # Increment age by 1

df['City'] = df['City'].map({'New York': 'NY', 'Chicago': 'CHI', 'Los Angeles': 'LA'})

df['City'].replace('NY', 'New York', inplace=True)
```

8. Grouping and Aggregation

Group data based on categories and perform aggregate operations.

Examples

Grouping:

```
grouped = df.groupby('City')
print(grouped['Age'].mean())  # Average age per city
```

Aggregation:

```
print(df.groupby('City').agg({'Age': ['min', 'max', 'mean']}))
```

9. Merging and Joining DataFrames

Combine multiple DataFrames using `merge`, `join`, and `concat`.

Examples

Merging:

```
df1 = pd.DataFrame({'Name': ['Alice', 'Bob'], 'Age': [25, 30]})
df2 = pd.DataFrame({'Name': ['Alice', 'Bob'], 'City': ['NY', 'CHI']})
merged = pd.merge(df1, df2, on='Name')
```

Concatenating:

```
df1 = pd.DataFrame({'A': [1, 2]})
df2 = pd.DataFrame({'A': [3, 4]})
concatenated = pd.concat([df1, df2], ignore_index=True)
```

10. Time Series Analysis

Pandas excels at handling time series data with functions for resampling and rolling statistics.

Examples

Creating Date Ranges:

```
dates = pd.date_range(start='2022-01-01', periods=5, freq='D')
```

Resampling:

```
df.set_index('Date', inplace=True)
resampled = df.resample('M').mean()  # Resample to monthly average
```

11. Statistical and Data Analysis

Pandas provides methods for descriptive statistics and correlations.

Examples

Statistical Summary:

```
print(df.describe())  # Basic statistical summary
```

Correlation:

```
print(df.corr())  # Correlation matrix
```

Value Counts:

```
print(df['City'].value_counts())  # Count occurrences of each unique value
```

12. Pivot Tables

Pivot tables are useful for summarizing and restructuring data.

Example

```
pivot_table = df.pivot_table(values='Age', index='City', columns='Gender', aggfunc='mean')
```

Summary of Important Pandas Methods

1. **Data Creation: `DataFrame, Series, read_csv, to_csv`**
2. **Data Inspection: `head, tail, info, describe, shape, columns`**
3. **Data Selection and Filtering: `loc, iloc`, conditional filtering**
4. **Data Cleaning: `dropna, fillna, drop_duplicates, rename`**
5. **Data Transformation: `apply, map, replace`**
6. **Grouping and Aggregation: `groupby, agg`**
7. **Merging and Joining: `merge, join, concat`**
8. **Time Series: `date_range, resample, rolling`**
9. **Statistical Analysis: `describe, corr, value_counts`**
10. **Pivot Tables: `pivot_table`**

CONCLUSION

Today, we explored two essential Python libraries: NumPy for efficient numerical operations and Pandas for data manipulation. NumPy helps with handling arrays and performing calculations quickly, while Pandas provides powerful tools for data cleaning and transformation. Together, these tools streamline data workflows, allowing us to analyze and manipulate data more effectively in data science and analytics tasks. Furthermore, I explored how to plot graphs using Matplotlib, which enhanced my ability to visualize data trends and communicate insights more effectively.