Numpy

August 31, 2024

1 getting familiarity

installing numpy

[3]:
```
pip install numpy
```

```
Defaulting to user installation because normal site-packages is not writeable
Requirement already satisfied: numpy in c:\users\lokesh
naidu\appdata\roaming\python\python312\site-packages (2.0.0)
Note: you may need to restart the kernel to use updated packages.


[notice] A new release of pip is available: 24.1.1 -> 24.2
[notice] To update, run: python.exe -m pip install --upgrade pip
```

importing numpy

[1]:
```
import numpy as np
```

array creation unsing numpy

creating 1D array

[42]:
```
a = np.array([1, 2, 3, 4])
print("1D array:", a)
```

```
1D array: [1 2 3 4]
```

creating 2D array

[43]:
```
b = np.array([[1, 2], [3, 4]])
print("2D array:", b)
```

```
2D array: [[1 2]
 [3 4]]
```

array with zeros

[44]:
```
c = np.zeros((2, 3))
print("Array of zeros:", c)
```

```
Array of zeros: [[0. 0. 0.]
 [0. 0. 0.]]
```

Array with a range of values

[45]:
```python
d = np.arange(0, 10, 2)
print("Array with a range:", d)
```

```
Array with a range: [0 2 4 6 8]
```

array properties 1.shape:dimentions of the array

[37]:
```python
print(a.shape)
print(b.shape)
```

```
(4,)
(2, 2)
```

2.sixe: total no of elements in that array

[38]:
```python
print(a.size)
print(b.size)
```

```
4
4
```

3.dtype: the data type of elements in the array

[41]:
```python
print(a.dtype)
print(b.dtype)
```

```
int64
int64
```

4.ndim: no of axes of the array

[40]:
```python
print(a.ndim)
print(b.ndim)
```

```
1
2
```

5. type:type of the array object

[39]:
```python
print(type(a))
```

```
<class 'numpy.ndarray'>
```

2 Python program to demonstrate data manipulation using Numpy.

array creation

[46]:
```python
arr1 = np.array([10, 20, 30, 40, 50])
print("1D Array:\n", arr1)
arr2 = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
```

```
print("2D Array:", arr2)
```

```
1D Array:
 [10 20 30 40 50]
2D Array: [[1 2 3]
 [4 5 6]
 [7 8 9]]
```

Indexing and Slicing Accessing elements

[11]:
```
element = arr2[1, 2]
print("Element at (1, 2):", element)
```

```
Element at (1, 2): 6
```

Slicing the 2D array

[47]:
```
slice_arr2 = arr2[:2, 1:]
print("Sliced Array:", slice_arr2)
```

```
Sliced Array: [[2 3]
 [5 6]]
```

Boolean indexing

[13]:
```
bool_index = arr1[arr1 > 25]
print("Boolean Indexing (values > 25):", bool_index)
```

```
Boolean Indexing (values > 25): [30 40 50]
```

reshaping arrays Reshape a 1D array to a 2D array

[14]:
```
reshaped_arr = arr1.reshape(1, 5)
print("Reshaped 1D Array to 2D:\n", reshaped_arr)
```

```
Reshaped 1D Array to 2D:
 [[10 20 30 40 50]]
```

Flattening a 2D array to a 1D array

[15]:
```
flattened_arr = arr2.flatten()
print("Flattened 2D Array:\n", flattened_arr)
```

```
Flattened 2D Array:
 [1 2 3 4 5 6 7 8 9]
```

mathematical operations element wise operations

[16]:
```
# Element-wise addition
arr_sum = arr2 + 10
print("Element-wise Addition:\n", arr_sum)

# Element-wise multiplication
```

```
arr_product = arr2 * 2
print("Element-wise Multiplication:\n", arr_product)

# Square root of each element
arr_sqrt = np.sqrt(arr2)
print("Square Root:\n", arr_sqrt)
```

```
Element-wise Addition:
 [[11 12 13]
 [14 15 16]
 [17 18 19]]
Element-wise Multiplication:
 [[ 2  4  6]
 [ 8 10 12]
 [14 16 18]]
Square Root:
 [[1.         1.41421356 1.73205081]
 [2.         2.23606798 2.44948974]
 [2.64575131 2.82842712 3.        ]]
```

3 data aggregation

1.Numpy functions to compute summary statistics like mean, median, standarddeviation, and sum.

[48]:
```
data=arr1
# Mean
mean = np.mean(data)
print("Mean:", mean)

# Median
median = np.median(data)
print("Median:", median)

# Standard Deviation
std_dev = np.std(data)
print("Standard Deviation:", std_dev)

# Sum
total_sum = np.sum(data)
print("Sum:", total_sum)
```

```
Mean: 30.0
Median: 30.0
Standard Deviation: 14.142135623730951
Sum: 150
```

2.Grouping Data and Performing Aggregations consider marks of students in three different subjects and grouping the marks based on subjects

[51]:
```python
scores = np.array([
    [85, 90, 78],
    [92, 88, 81],
    [70, 95, 80],
    [88, 76, 85],
    [90, 85, 88],
    [74, 80, 77]
])

# Define subject names
subjects = ['python', 'java', 'c']

print("Scores Data:\n", scores)
print("\n")

# Grouping data by subject and performing aggregations
print("Grouping Data and Aggregations:")
for i, s in enumerate(subjects):
    subject_scores = scores[:, i]

    subject_mean = np.mean(subject_scores)
    subject_median = np.median(subject_scores)
    subject_std = np.std(subject_scores)
    subject_sum = np.sum(subject_scores)

    print(f"Subject: {s}")
    print(f"  Mean Score: {subject_mean:}")
    print(f"  Median Score: {subject_median:}")
    print(f"  Standard Deviation of Scores: {subject_std:}")
    print(f"  Total Sum of Scores: {subject_sum:}")
```

```
Scores Data:
 [[85 90 78]
 [92 88 81]
 [70 95 80]
 [88 76 85]
 [90 85 88]
 [74 80 77]]


Grouping Data and Aggregations:
Subject: python
  Mean Score: 83.16666666666667
  Median Score: 86.5
  Standard Deviation of Scores: 8.254628331359863
  Total Sum of Scores: 499
Subject: java
  Mean Score: 85.66666666666667
  Median Score: 86.5
  Standard Deviation of Scores: 6.289320754704403
  Total Sum of Scores: 514
Subject: c
  Mean Score: 81.5
  Median Score: 80.5
  Standard Deviation of Scores: 3.8622100754188224
  Total Sum of Scores: 489
```

4 data analysis

1.finding correlation

[25]:
```
correlation_matrix = np.corrcoef(scores.T)
print("Correlation Matrix:\n", correlation_matrix)
```

```
Correlation Matrix:
 [[ 1.         -0.30069939  0.58289483]
 [-0.30069939  1.         -0.29503774]
 [ 0.58289483 -0.29503774  1.        ]]
```

2.identifying outliers based on z-scores

[29]:
```
for i, s in enumerate(subjects):
    subject_scores = scores[:, i]

    mean = np.mean(subject_scores)
    std_dev = np.std(subject_scores)

    z_scores = (subject_scores - mean) / std_dev
    print(f"z_scores:{z_scores}")
    outliers = subject_scores[np.abs(z_scores) > 1.96]

    print(f"Subject: {s}")
    print(f"  Outliers (if any): {outliers}")
```

```
z_scores:[ 0.22209762  1.07010673 -1.59506475  0.5855301   0.82781841
-1.11048812]
Subject: python
  Outliers (if any): []
z_scores:[ 0.68899862  0.37099926  1.48399703 -1.53699693 -0.10599979 -0.9009982
]
Subject: java
  Outliers (if any): []
z_scores:[-0.90621689 -0.12945956 -0.38837867  0.90621689  1.68297422 -1.165136
]
Subject: c
  Outliers (if any): []
```

3.Calculating Percentiles:

```
[53]: for i, s in enumerate(subjects):
          subject_scores = scores[:, i]
          percentiles = np.percentile(subject_scores, [25, 50, 75])
          print(f"Subject: {s}")
          print(f"  25th Percentile: {percentiles[0]:}")
          print(f"  50th Percentile (Median): {percentiles[1]:}")
          print(f"  75th Percentile: {percentiles[2]:}")
```

```
Subject: python
  25th Percentile: 76.75
  50th Percentile (Median): 86.5
  75th Percentile: 89.5
Subject: java
  25th Percentile: 81.25
  50th Percentile (Median): 86.5
  75th Percentile: 89.5
Subject: c
  25th Percentile: 78.5
  50th Percentile (Median): 80.5
  75th Percentile: 84.0
```

5 conclusion

NumPy was used to perform various data analysis tasks including aggregation, correlation calculation, outlier detection, and percentile computation.

A)Efficiency in Data Handling and Computation:

Vectorized Operations: NumPy enables vectorized operations which apply functions to entire arrays at once. This approach avoids the need for explicit loops in Python, leading to faster execution.

Memory Efficiency: NumPy arrays consume less memory compared to Python lists due to their fixed-size data types.

B)Advanced Mathematical and Statistical Functions:

Built-in Functions: NumPy provides a comprehensive set of mathematical and statistical functions such as np.mean, np.std, np.corrcoef, and np.percentile. These functions are optimized for performance and can handle large-scale data computations efficiently.

Aggregation and Analysis: Functions for aggregation and statistical analysis in NumPy are designed to process multi-dimensional arrays and perform complex calculations quickly, making data analysis tasks simpler and more efficient.

6 Advantages of NumPy Over Traditional Python Data Structures

Performance:

Speed: NumPy operations are executed in compiled code (C and Fortran), making them much faster than operations performed with Python's built-in data structures. For instance, operations on NumPy arrays are often several orders of magnitude faster than equivalent operations on lists or tuples.

Memory Management:

Compact Storage: NumPy arrays are more memory-efficient compared to Python lists. They use a contiguous block of memory, which reduces overhead and improves cache performance, making them suitable for large datasets.

Ease of Use:

Conciseness: NumPy simplifies code by allowing complex numerical operations to be expressed in a few lines of code. This results in more readable and maintainable code, especially when dealing with large datasets and complex analyses.

Rich Functionality:

Extensive Libraries: NumPy integrates well with other scientific libraries such as SciPy, pandas, and scikit-learn. This makes it a cornerstone of the scientific computing stack in Python, facilitating data manipulation, analysis, and machine learning tasks.

7 Real-World Examples Where NumPy's Capabilities Are Crucial

1.Machine Learning:

Data Preprocessing: NumPy is used extensively for preprocessing data, including normalization, scaling, and transforming datasets. Libraries like scikit-learn rely on NumPy for efficient numerical computations.

Model Training: NumPy arrays are used to represent feature matrices and target vectors. Efficient computations with NumPy are crucial for training machine learning models on large datasets.

2.Financial Analysis:

Risk Management: NumPy is used to calculate financial metrics such as volatility, Value at Risk (VaR), and returns. Its ability to handle large arrays efficiently is essential for analyzing time series data and running simulations.

Portfolio Optimization: NumPy facilitates the optimization of financial portfolios by computing returns, covariances, and efficient frontiers. It handles large matrices involved in optimization problems effectively.

3.Scientific Research:

Numerical Simulations: In scientific research, NumPy is used for simulations in physics, chemistry, and biology. Its efficient handling of large arrays and matrix operations is vital for simulating complex systems.

Data Analysis: Scientists use NumPy to analyze experimental data, perform statistical tests, and visualize results. The library's capabilities enable the handling of large-scale datasets and complex mathematical computations.

```
[ ]:
```