numpy-in-ds

September 1, 2024

#NUMPY NumPy is the fundamental package for scientific computing in Python. It is a Python library that provides a multidimensional array object, various derived objects (such as masked arrays and matrices), and an assortment of routines for fast operations on arrays, including mathematical, logical, shape manipulation, sorting, selecting, I/O, discrete Fourier transforms, basic linear algebra, basic statistical operations, random simulation and much more.

1. Getting Familiar with Numpy:

[1]:
```
#importing numpy package
import numpy as np
```

[3]:
```
#creating a 1D array
a = np.array([1, 2, 3, 4])
a
```

[3]:
```
array([1, 2, 3, 4])
```

[5]:
```
#creating ND array by generating some random data
b=np.random.randn(3,3)
b
```

[5]:
```
array([[ 1.78774452,  2.7067487 ,  0.71805876],
       [ 0.9361449 , -0.69975256,  0.30583089],
       [ 0.44712137,  0.78558158, -0.79341392]])
```

[12]:
```
#creating numpy array using other functions
zero_arr=np.zeros((3, 2))
ones_arr=np.ones((3,3))
emp_arr=np.empty((2, 3))
print("zero array is:",zero_arr)
print("ones array is:",ones_arr)
print("empty array",emp_arr)
```

```
zero array is: [[0. 0.]
 [0. 0.]
 [0. 0.]]
ones array is: [[1. 1. 1.]
 [1. 1. 1.]
 [1. 1. 1.]]
```

```
empty array [[0. 0. 0.]
 [0. 0. 0.]]
```

[13]:
```
#Numpy properties
print(b.ndim)
print(b.shape)
print(b.dtype)
print(b.size)
```

```
2
(3, 3)
float64
9
```

2.DATA MANIPULATION

[17]:
```
#INDEXING AND SLICING
arr = np.arange(10)
print(arr[5])
print(arr[5:8])
arr[5:8] = 12
print(arr)
```

```
5
[5 6 7]
[ 0  1  2  3  4 12 12 12  8  9]
```

[18]:
```
#ARRAY SLICING (BROADCASTING PROPERTY OF NUMPY)
arr_slice = arr[5:8]
print(arr_slice)
arr_slice[1] = 12345
print(arr)
```

```
[12 12 12]
[    0     1     2     3     4    12 12345    12     8     9]
```

[4]:
```
#INDEXING AND SLICING FOR ND ARRAY
import numpy as np
arr2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
print(arr2d[2])
print(arr2d[0][2])
#Accessing first 2 rows of nd array
print(arr2d[:2])
print(arr2d[:2, 1:])
arr2d[:2, 1:] = 0
print(arr2d)
```

```
[7 8 9]
3
```

```
[[1 2 3]
 [4 5 6]]
[[2 3]
 [5 6]]
[[1 0 0]
 [4 0 0]
 [7 8 9]]
```

RESHAPING:

```
[5]: arr = np.arange(15).reshape((3, 5))
     print(arr)
```

```
[[ 0  1  2  3  4]
 [ 5  6  7  8  9]
 [10 11 12 13 14]]
```

```
[6]: #Transpose array
     print(arr.T)
```

```
[[ 0  5 10]
 [ 1  6 11]
 [ 2  7 12]
 [ 3  8 13]
 [ 4  9 14]]
```

Mathematical operations

```
[11]: #ADDITION
      arr+arr
```

```
[11]: array([[ 0,  2,  4,  6,  8],
             [10, 12, 14, 16, 18],
             [20, 22, 24, 26, 28]])
```

```
[9]: #SUBTRACTION
     arr-arr
```

```
[9]: array([[0, 0, 0, 0, 0],
            [0, 0, 0, 0, 0],
            [0, 0, 0, 0, 0]])
```

```
[10]: #MULTIPLICATION
      arr*arr
```

```
[10]: array([[  0,   1,   4,   9,  16],
             [ 25,  36,  49,  64,  81],
             [100, 121, 144, 169, 196]])
```

[13]:
```
#DIVISION
arr/2
```

[13]:
```
array([[0. , 0.5, 1. , 1.5, 2. ],
       [2.5, 3. , 3.5, 4. , 4.5],
       [5. , 5.5, 6. , 6.5, 7. ]])
```

3.Data Aggregation

[15]:
```
c=np.random.randint(1,50,5)
c
```

[15]:
```
array([ 2,  7, 23, 23, 35])
```

[16]:
```
mean=np.mean(c)
mean
```

[16]:
```
18.0
```

[18]:
```
median=np.median(c)
median
```

[18]:
```
23.0
```

[20]:
```
#Standard deviation
std=np.std(c)
std
```

[20]:
```
11.966620241321273
```

[21]:
```
product=np.prod(c)
product
```

[21]:
```
259210
```

[22]:
```
sum=np.sum(c)
sum
```

[22]:
```
90
```

4.Data Analysis

Correlation:Correlation measures the strength and direction of the linear relationship between two variables. NumPy provides tools for computing correlation coefficients.

[24]:
```
import numpy as np

data1 = np.array([1, 2, 3, 4, 5])
```

```
data2 = np.array([2, 3, 4, 5, 6])

# Compute correlation coefficient
correlation = np.corrcoef(data1, data2)[0, 1]
print(f"Correlation coefficient: {correlation}")
```

```
Correlation coefficient: 0.9999999999999999
```

2. Identifying Outliers Outliers are data points that differ significantly from other observations. One common method to identify outliers is using the Z-score or Interquartile Range (IQR). We'll use the Z-score method here.

[33]:
```
data = np.random.randint(100,1000,10)
mean=np.mean(data)
std=np.std(data)
# Compute Z-scores
z_scores = (data-mean)/std
print(f"Z-scores: {z_scores}")

# Identify outliers (typically Z-score > 3 or < -3)
outliers = np.where(np.abs(z_scores) > 3)
print(f"Outliers: {data[outliers]}")
```

```
Z-scores: [ 1.43097393 -1.34365072 -0.51267176 -0.27323715  0.12582054
-1.20750162
  1.22440289  1.36055198  0.17276851 -0.9774566 ]
Outliers: []
```

3. Calculating Percentiles Percentiles are used to understand the distribution of data. For instance, the 25th percentile (Q1) is the value below which 25% of the data falls.

[34]:
```
data = np.array([1, 3, 5, 7, 9, 11, 13, 15])

# Calculate percentiles
percentile_25 = np.percentile(data, 25)
percentile_50 = np.percentile(data, 50)  # This is also the median
percentile_75 = np.percentile(data, 75)

print(f"25th percentile: {percentile_25}")
print(f"50th percentile (median): {percentile_50}")
print(f"75th percentile: {percentile_75}")
```

```
25th percentile: 4.5
50th percentile (median): 8.0
75th percentile: 11.5
```

Efficiency of NumPy NumPy is highly efficient for large datasets due to its underlying implementation in C and optimized data structures. Its array operations are vectorized, meaning that operations are performed in bulk rather than through explicit loops, which leads to significant performance gains.

[40]:
```python
import time


# Large dataset
large_data = np.random.rand(1000000)

# Time vectorized operation-numpy
start_time = time.time()
mean_large_data = np.mean(large_data)
end_time = time.time()
print(f"Mean of large data (vectorized): {mean_large_data}")
print(f"Time taken (vectorized): {end_time - start_time:.6f} seconds")

# Time non-vectorized operation-python loop for comparing
start_time = time.time()
mean_large_data_non_vec = np.sum(large_data) / len(large_data)
end_time = time.time()
print(f"Mean of large data (non-vectorized):{mean_large_data_non_vec}")
print(f"Time taken (non-vectorized): {end_time - start_time:.6f} seconds")
```

```
Mean of large data (vectorized): 0.500056611322885
Time taken (vectorized): 0.016804 seconds
Mean of large data (non-vectorized):0.500056611322885
Time taken (non-vectorized): 0.002299 seconds
```

Conclusion: The Role of NumPy in Data Science

In data science, numerical computations and data manipulation are fundamental tasks. NumPy, a powerful library in Python, plays a crucial role in making these tasks more efficient and effective. Let's explore how NumPy can benefit data science professionals and its advantages over traditional Python data structures. - Speed: NumPy arrays (ndarrays) are implemented in C, which means they are much faster than Python's built-in lists and tuples when it comes to numerical operations. This speed is crucial for handling large datasets and performing complex computations efficiently. - Memory Efficiency: NumPy arrays use less memory compared to Python lists. They store data in contiguous blocks of memory, which minimizes overhead and reduces the memory footprint of your data. - Vectorized Operations: NumPy supports vectorized operations, allowing you to perform operations on entire arrays of data without the need for explicit loops. This leads to cleaner, more readable code and significantly faster execution. NumPy includes a wide range of mathematical functions and operations (e.g., linear algebra, statistics, Fourier transforms) that are optimized for performance and designed to handle large datasets effectively. - NumPy provides advanced indexing and slicing capabilities that allow for more complex data manipulations compared to Python lists. This includes boolean indexing, multi-dimensional slicing, and array broadcasting. - NumPy integrates seamlessly with other scientific computing libraries, such as Pandas, SciPy, and scikit-learn. This integration is vital for building end-to-end data science workflows.

Real-World Examples of NumPy's Capabilities

1. Machine Learning:
    - Data Preprocessing:In machine learning, data preprocessing often involves operations

like normalization, scaling, and transformation. NumPy's vectorized operations make these tasks efficient and straightforward. For example, when normalizing features to a common scale, NumPy can apply mathematical transformations to entire arrays in a single line of code.
    - Algorithm Implementation: Many machine learning algorithms, including those for classification, regression, and clustering, rely on linear algebra and numerical computations. NumPy provides the necessary functions to implement these algorithms efficiently. For instance, matrix operations involved in neural networks or support vector machines can be handled with NumPy's linear algebra functions.
2. Financial Analysis:
    - Quantitative Analysis:In finance, tasks such as risk assessment, portfolio optimization, and option pricing involve complex numerical computations. NumPy's ability to handle large arrays and perform fast mathematical operations makes it an essential tool for these analyses. For instance, calculating the moving average of stock prices or simulating financial models can be done efficiently using NumPy.
    - Time Series Analysis:Financial data often comes in time series format, which requires specialized methods for analysis. NumPy helps in managing and analyzing time series data by providing functions for statistical operations, data transformations, and calculations of moving averages.
3. Scientific Research:
    - Data Analysis and Visualization: In scientific research, handling large datasets and performing numerical simulations are common. NumPy's array operations are used to process experimental data, perform statistical analysis, and generate visualizations. For example, in physics, NumPy can be used to analyze data from experiments or simulations, perform Fourier transforms for signal processing, and solve differential equations.
    - Simulation and Modeling:Scientific simulations, such as those in biology or climate science, often involve large-scale computations. NumPy provides the performance needed to run these simulations efficiently, enabling researchers to model complex systems and analyze results.

SUMMARY In summary, NumPy is an indispensable tool for data science professionals due to its speed, efficiency, and rich set of features for numerical computations. Its advantages over traditional Python data structures, such as lists and tuples, make it essential for handling large datasets and performing complex mathematical operations across various domains including machine learning, financial analysis, and scientific research.

```
[ ]:
```