untitled5

September 1, 2024

1 NUMPY

NumPy (Numerical Python) is a powerful Python library for numerical computing. It provides support for large, multi-dimensional arrays and matrices, along with a collection of high-level mathematical functions to operate on these arrays efficiently. NumPy is a fundamental package for scientific computing and data analysis in Python and is often used as the foundation for other libraries like SciPy, Pandas, and Tensorflow.

1.1 GETTING FAMILIAR WITH NUMPY

Array Creation:

[117]:
```python
#importing numpy
import numpy as np
```

[107]:
```python
#From a python list or tuple
arr=np.array([1,2,3,4])
arr
```

[107]:
```
array([1, 2, 3, 4])
```

[56]:
```python
#Creating a 2D array(nd array)
arr_2d=np.array([[1,2,3],[4,5,6]])
arr_2d
```

[56]:
```
array([[1, 2, 3],
       [4, 5, 6]])
```

[54]:
```python
#zeroes,ones,empty
arr=np.zeros((2,2))
print(arr)
arr1=np.ones((3,3))
print(arr1)
arr2=np.empty((2,2))
print(arr2)
```

```
[[0. 0.]
 [0. 0.]]
[[1. 1. 1.]
 [1. 1. 1.]
 [1. 1. 1.]]
[[0. 0.]
 [0. 0.]]
```

[10]:
```
#Creating array with ranges
range_arr = np.arange(10)      # Array of values from 0 to 9
print(range_arr)
```

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

[52]:
```
arr=np.linspace(0,1,5)
arr
```

[52]:
```
array([0.  , 0.25, 0.5 , 0.75, 1.  ])
```

[15]:
```
#Creating array with random values
random_arr=np.random.rand(5)    #1d array
print(random_arr)
```

```
[0.67882333 0.16240219 0.87876441 0.05815489 0.79033178]
```

[16]:
```
arr_2d=np.random.rand(3,4)    #2d array
print(arr_2d)
```

```
[[0.52523642 0.75812504 0.90531862 0.26482215]
 [0.73391497 0.61756809 0.35300605 0.78886522]
 [0.56313038 0.86918053 0.84719352 0.76094303]]
```

Basic operations :

[36]:
```
#Arithmetic operations
a=np.array([1,2,3])
b=np.array([4,5,6])
print(a+b)   #addition
```

```
[5 7 9]
```

[21]:
```
print(a-b)    #subtraction
```

```
[-3 -3 -3]
```

[22]:
```
print(a*b)     #multiplication
```

```
[ 4 10 18]
```

[23]:
```
print(a/b)      #Division
```

```
[0.25 0.4  0.5 ]
```

[25]:
```python
#Mathematical function
sqrt_arr = np.sqrt(a)  # Square root of each element
print(sqrt_arr)
```

```
[1.         1.41421356 1.73205081]
```

[51]:
```python
#Aggregation
array = np.array([1, 2, 3, 4, 5])
print(np.sum(array))   # 15
print(np.mean(array))  # 3.0
print(np.max(array))  # 5
print(np.min(array))
```

```
15
3.0
5
1
```

Properties of an array:

[49]:
```python
array=np.array([[1,2,3],[4,5,6]])
#To find the number of dimesions(axes) of the array
array.ndim
```

[49]:
```
2
```

[48]:
```python
#To find total no.of elements in the array
array.size
```

[48]:
```
6
```

[30]:
```python
#To find datatype of an array
print(array.dtype)
```

```
int64
```

[46]:
```python
#To find the size of each element
array.itemsize
```

[46]:
```
8
```

[47]:
```python
#To find shape
array.shape     #2rows and 3columns
```

[47]:
```
(2, 3)
```

[ ]:
```python
#Example for understand all properties
import numpy as np
```

```
array = np.array([[1, 2, 3], [4, 5, 6]])

print("Shape:", array.shape)
print("Number of dimensions:", array.ndim)
print("Size (number of elements):", array.size)
print("Data type:", array.dtype)
print("Item size (bytes):", array.itemsize)
print("Total bytes consumed:", array.nbytes)
print("Strides:", array.strides)
print("Array flags:\n", array.flags)
```

1.2 DATA MANIPULATION

1.2.1 Slicing:

Slicing in NumPy is used to extract a portion of an array. It allows you to access and manipulate specific subsets of data within an array. Here's a basic overview of how slicing works

1d array

[59]:
```
#creating an array
arr=np.arange(20)
arr
```

[59]:
```
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16,
       17, 18, 19])
```

[60]:
```
#extracting a part of array
arr[3:9]
```

[60]:
```
array([3, 4, 5, 6, 7, 8])
```

2d array

[62]:
```
arr_2d=np.array([[1,2,3],[4,5,6],[7,8,9]])
arr_2d
```

[62]:
```
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
```

[63]:
```
#slicing a subarray
arr_2d[0:2,1:3]
```

[63]:
```
array([[2, 3],
       [5, 6]])
```

[64]:
```
arr_2d[:4,1:]
```

```
[64]: array([[2, 3],
             [5, 6],
             [8, 9]])
```

1.2.2 Reshaping:

In NumPy, reshaping is a common operation used to change the shape of an array without altering its data. You can use the reshape method or the reshape function to do this. Here's how

```
[67]: arr=np.array([1,2,3,4,5,6])
      reshaped_arr=arr.reshape((2,3))
      reshaped_arr
```

```
[67]: array([[1, 2, 3],
             [4, 5, 6]])
```

the new shape must be compatible with the original number of elements. For instance, if the original array has 6 elements, it can be reshaped into any shape where the product of dimensions equals 6 (e.g., 2x3, 3x2, etc.).If you try to reshape an array into a shape where the number of elements doesn't match, NumPy will raise a ValueError

1.2.3 Applying mathematical operations:

```
[85]: data=np.arange(20)
      data
```

```
[85]: array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16,
             17, 18, 19])
```

```
[91]: data1=data.reshape((5,4))
      data1
```

```
[91]: array([[ 0,  1,  2,  3],
             [ 4,  5,  6,  7],
             [ 8,  9, 10, 11],
             [12, 13, 14, 15],
             [16, 17, 18, 19]])
```

```
[90]: #Addition of 2 Arrays
      data1=data+data
      data1
```

```
[90]: array([ 0,  2,  4,  6,  8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32,
             34, 36, 38])
```

```
[87]: #Multiplication
      data1=data*data
      data1
```

```
[87]: array([  0,   1,   4,   9,  16,  25,  36,  49,  64,  81, 100, 121, 144,
              169, 196, 225, 256, 289, 324, 361])
```

```
[115]: #subtraction
       data2=data1-data1        #Here data1 means reshaped array(5,4)
       data2
```

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

```
[116]: #square root
       np.sqrt(data1)          #data1=(4,6) reshaped array
```

```
[116]: array([[0.        , 1.        , 1.41421356, 1.73205081],
              [2.        , 2.23606798, 2.44948974, 2.64575131],
              [2.82842712, 3.        , 3.16227766, 3.31662479],
              [3.46410162, 3.60555128, 3.74165739, 3.87298335],
              [4.        , 4.12310563, 4.24264069, 4.35889894]])
```

1.3 **DATA AGGREGATION**

Data aggregation in NumPy involves summarizing data using functions that compute statistics or aggregate values over specified axes of an array. Here are some common aggregation functions and techniques

```
[118]: arr=np.random.randint(1,50,5)
       arr
```

```
[118]: array([19, 33, 42, 49, 15], dtype=int32)
```

```
[119]: #To calculate sum
       sum=np.sum(arr)
       sum
```

```
[119]: np.int64(158)
```

```
[120]: #To calculate mean
       mean=np.mean(arr)
       mean
```

```
[120]: np.float64(31.6)
```

```
[121]: #To calculate median
       median=np.median(arr)
       median
```

[121]: np.float64(33.0)

[122]:
```
#To calculate standard deviation
std=np.std(arr)
std
```

[122]: np.float64(13.016912076218384)

1.4 DATA ANALYSIS

1.4.1 Correlation:

`To find correlations between two datasets,you can use Numpy's np.corrcoef function`

[123]:
```
import numpy as np

# Example datasets
x = np.array([1, 2, 3, 4, 5])
y = np.array([2, 3, 4, 5, 6])

# Calculate correlation coefficient matrix
correlation_matrix = np.corrcoef(x, y)
print("Correlation Matrix:\n", correlation_matrix)

# Extract correlation coefficient between x and y
correlation_xy = correlation_matrix[0, 1]
print("Correlation Coefficient between x and y:", correlation_xy)
```

```
Correlation Matrix:
 [[1. 1.]
 [1. 1.]]
Correlation Coefficient between x and y: 0.9999999999999999
```

1.4.2 Identify outliers :

In NumPy, the numpy library doesn't have a built-in function specifically for detecting outliers, but you can perform outlier detection using statistical methods such as the Interquartile Range (IQR) method.

Here's how you can identify outliers using NumPy:

1. *Calculate the IQR:*
   - Find the first quartile (Q1) and third quartile (Q3) of your data.
   - Compute the IQR as Q3 - Q1.
2. *Determine the outlier thresholds:*
   - Define the lower bound as Q1 - 1.5 * IQR.
   - Define the upper bound as Q3 + 1.5 * IQR.
3. *Identify outliers:*
   - Any data points outside these bounds are considered outliers

[124]:
```python
import numpy as np

# Sample data
data = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 100])

# Compute Q1 (25th percentile) and Q3 (75th percentile)
Q1 = np.percentile(data, 25)
Q3 = np.percentile(data, 75)

# Calculate the IQR
IQR = Q3 - Q1

# Determine the bounds for outliers
lower_bound = Q1 - 1.5 * IQR
upper_bound = Q3 + 1.5 * IQR

# Identify outliers
outliers = data[(data < lower_bound) | (data > upper_bound)]

print("Outliers:", outliers)
```

```
Outliers: [100]
```

1.4.3 Efficiency in handling large datasets

NumPy is highly efficient for handling large datasets due to: 1.Vectorized Operations: NumPy performs operations on entire arrays at once, leveraging optimized low-level implementations. 2.Memory Efficiency: NumPy arrays are stored in contiguous blocks of memory, which reduces overhead compared to Python lists. 3.Broadcasting: NumPy supports broadcasting, which allows for efficient computation without explicit loops. 4.Integration with C and Fortran Libraries: NumPy can interface with optimized C and Fortran libraries for performance-intensive operations.

[125]:
```python
import numpy as np

# Create a large dataset
large_data = np.random.randn(10**7)

# Calculate mean and standard deviation efficiently
mean = np.mean(large_data)
std_dev = np.std(large_data)

print("Mean:", mean)
print("Standard Deviation:", std_dev)
```

```
Mean: -0.0003032763910120176
Standard Deviation: 1.0003908036410256
```

1.5 **Application in Data Science**

NumPy is a foundational tool for data science due to its powerful capabilities for numerical computations. Here's a summary of how NumPy benefits data science professionals and its advantages over traditional Python data structures:

Advantages of NumPy for Data Science

1. *Performance*: NumPy arrays are implemented in C, providing significant performance improvements over Python lists. Operations on NumPy arrays are executed at compiled speeds, which is crucial for handling large datasets and complex calculations efficiently.
2. *Vectorization*: NumPy allows for vectorized operations, enabling data scientists to perform mathematical operations on entire arrays at once without explicit loops. This leads to more concise and faster code.
3. *Memory Efficiency*: NumPy arrays consume less memory than Python lists due to their fixed size and homogeneous data types. This is important when working with large datasets.
4. *Broad Functionality*: NumPy provides a comprehensive set of mathematical, statistical, and linear algebra functions, making it a one-stop solution for many numerical computation tasks.
5. *Integration*: NumPy integrates well with other libraries such as pandas for data manipulation, SciPy for scientific computations, and scikit-learn for machine learning, creating a robust ecosystem for data analysis and modeling.

Real-World Examples

1. *Machine Learning*:
    - *Data Preprocessing*: NumPy is used to preprocess data by performing operations like normalization, standardization, and feature extraction efficiently.
    - *Model Training*: Libraries like scikit-learn use NumPy arrays to represent datasets, making it possible to apply machine learning algorithms on large-scale data quickly.

    Example: Training a neural network often involves matrix operations and optimizations that are efficiently handled by NumPy. For instance, the weight updates in gradient descent algorithms are performed using NumPy operations.
2. *Financial Analysis*:
    - *Portfolio Optimization*: NumPy can be used to calculate returns, risks, and correlations of financial assets, aiding in the optimization of investment portfolios.
    - *Risk Management*: It helps in computing Value at Risk (VaR) and other risk metrics by performing statistical analyses on large financial datasets.

    Example: Calculating the historical volatility of a stock requires operations on large arrays of price data, which can be efficiently performed using NumPy.
3. *Scientific Research*:
    - *Data Simulation*: Researchers use NumPy to generate random samples and perform simulations, which are crucial for hypothesis testing and modeling scientific phenomena.
    - *Statistical Analysis*: NumPy's statistical functions are used to analyze experimental data, fit models, and validate scientific theories.

    Example: In physics, simulations of particle movements involve large-scale computations on arrays, which are managed efficiently with NumPy.

0.0.1 Conclusion

NumPy provides a powerful, efficient, and versatile tool for data science professionals. Its ability to handle large datasets, perform complex numerical computations, and integrate with other data science libraries makes it indispensable. By leveraging NumPy, data scientists can achieve faster computation, reduced memory usage, and more readable code compared to using traditional Python data structures, ultimately enabling more effective and efficient analysis of large and complex datasets.