numpy

August 31, 2024

NUMPY

BASIC OPERATIONS USING NUMPY.

Creating a 2D Numpy Array:

Purpose: To create a 2-dimensional array (matrix) with specified values. Details: The array is initialized with two rows and five columns. The values are explicitly defined, creating a matrix where the first row contains numbers from 1 to 5, and the second row contains numbers from 6 to 10. Output: A matrix structure where elements are organized in rows and columns. Array of Zeros:

Purpose: To create a 2-dimensional array where all elements are zeros. Details: The array has dimensions 3x3, meaning it has three rows and three columns. Each element of this array is set to zero, resulting in a matrix filled entirely with zeros. Output: A 3x3 matrix where every element is zero, useful for initialization purposes where you want to start with a blank slate. Array of Ones:

Purpose: To create a 2-dimensional array where all elements are ones. Details: Similar to the zero array, this array also has dimensions 3x3. Each element in this array is set to one, creating a matrix where every entry is the number one. Output: A 3x3 matrix with all elements equal to one, which can be useful in various computational scenarios where you need a matrix with uniform values. Array with a Specific Value:

Purpose: To create a 2-dimensional array where all elements are set to a particular value. Details: The array is 3x3, and each element is initialized to a specific value (e.g., 4). This operation is useful when you need a matrix where all elements have the same predefined value. Output: A 3x3 matrix where every element is the specified value, which can be helpful for certain algorithms or data manipulations. Array with a Range of Values:

Purpose: To create a 1-dimensional array with values in a specified range. Details: The array is created with values starting from 0 up to, but not including, 20, with a step of 2 between consecutive values. This results in a sequence of numbers following the given step size. Output: A 1-dimensional array that contains numbers in the specified range, which is useful for generating sequences or ranges of values for various applications. Array with Random Values:

Purpose: To create a 2-dimensional array filled with random floating-point numbers. Details: The array has dimensions 2x2, and the values are randomly generated between 0 and 1. Each time the array is created, the values will differ due to their random nature. Output: A 2x2 matrix where each element is a random number between 0 and 1, useful for simulations, testing, or any application requiring random data.

These operations demonstrate various ways to initialize and work with arrays in Numpy, showcasing its capabilities for creating arrays with specific values, ranges, or random numbers, and for setting

up arrays for further computations.

The sample codes for the above contex is mentioned below.

[3]:
```
import numpy as np
#creating a numpy array(2_D array)
array = np.array([[1,2,3,4,5],[6,7,8,9,10],[11,12,13,14,15]])
print(array)

#some basic operations of numpy array
zero_arr = np.zeros((3,3)) #2_D array with only zeros
print("\n")
print("zero array :")
print(zero_arr)


one_arr = np.ones((3,3))#2_D array with only ones
print("\n")
print("ones array : ")
print(one_arr)

val_arr = np.full((3,3),4)#creates an array with specific value(here the value␣
 ↪is 4 and the dimensions are 3x3
print("\n")
print("array with specific value : ")
print(val_arr)


arr_range = np.arange(0,20,2)#creates an array within a range(the values␣
 ↪provided in the paranthesis indicates start,stop,step.It creates a one␣
 ↪dimensional array
print("\n")
print("Range array : ")
print(arr_range)


rand_arr = np.random.random((2,2))#Creates an array with random values with the␣
 ↪mentioned dimension in the paranthesis
print("\n")
print("Random array : ")
print(rand_arr)
```

```
[[ 1  2  3  4  5]
 [ 6  7  8  9 10]
 [11 12 13 14 15]]


zero array :
```

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


ones array :
[[1. 1. 1.]
 [1. 1. 1.]
 [1. 1. 1.]]


array with specific value :
[[4 4 4]
 [4 4 4]
 [4 4 4]]


Range array :
[ 0  2  4  6  8 10 12 14 16 18]


Random array :
[[0.94215375 0.61802822]
 [0.50041992 0.83535515]]
```

DATA MANIPULATION USING NUMPY

1. Array Slicing Array slicing in Numpy allows you to extract specific sections or sub-arrays from a larger array. In the provided code:

Slicing is used to select a portion of a 2D array. The slicing operation array[0:3, 1:4] extracts a sub-array from the original array. Rows: The slice 0:3 selects rows starting from index 0 up to, but not including, index 3. Columns: The slice 1:4 selects columns starting from index 1 up to, but not including, index 4. This operation effectively extracts a sub-array that spans from the first to the third row and from the second to the fourth column of the original array. 2. Array Reshaping Reshaping is a technique used to change the dimensions of an array while keeping its total number of elements constant.

In the example provided, the operation array.reshape(5, 3) changes the shape of the original array from 3x5 (3 rows and 5 columns) to 5x3 (5 rows and 3 columns). Reshape: It reorganizes the data from the original array into a new shape without altering the actual data content. The total number of elements remains the same. 3. Array Indexing Indexing is used to access individual elements or specific positions within an array.

The operation array[1, 3] accesses the element located at the second row (index 1) and the fourth column (index 3) of the array. Element Extraction: This specific index operation retrieves the value at the given position, allowing for targeted data retrieval within the array. 4. Basic Mathematical Operations Numpy supports a range of mathematical operations that are performed element-wise on arrays. These operations include:

Element-Wise Addition: The operation np.add(array, arr_2) adds corresponding elements from

two arrays. Each element in the resulting array is the sum of the elements at the same position in the original arrays.

Element-Wise Subtraction: The operation np.subtract(array, arr_2) subtracts corresponding elements in one array from those in another. The result is an array where each element is the difference between the elements at the same positions in the original arrays.

Element-Wise Multiplication: The operation np.multiply(array, arr_2) multiplies corresponding elements from two arrays. Each element in the resulting array is the product of the elements at the same position in the original arrays.

Element-Wise Division: The operation np.divide(array, arr_2) divides corresponding elements in one array by those in another. The resulting array contains the quotient of the division of elements at corresponding positions in the original arrays.

Element-Wise Power: The operation np.power(array, 2) raises each element in the array to the power of 2. This is used to compute the square of each element in the array.

Element-Wise Square Root: The operation np.sqrt(array) computes the square root of each element in the array. This is used to transform each element into its square root.

THE SAMPLE CODES ARE MENTIONED BELOW.

[30]:
```
#DATA MANIPULATION USING NUMPY
#ARRAY SLICING
print("the sliced array is : ")
print(array[0:3,1:4])#Slices the 0th column and 5th column of the 2_D array

#ARRAY RESHAPING
print("Reshaped array is : ")
reshaped_array = array.reshape(5,3)#reshapes the 3x5 array into 5x3 array
print(reshaped_array)

#ARRAY INDEXING
element = array[1,3]#stores the element at the mentioned position
print("the element at the position (1,3) in the array is : ", element)#prints␣
 ↪the element value

#BASIC MATHEMATICAL OPERARTIONS IN NUMPY
arr_2 = np.array([[11,12,13,14,15],[16,17,18,19,20],[21,22,23,24,25]])
add = np.add(array,arr_2) # Element-wise addition
difference = np.subtract(array,arr_2) #Element-wise subtraction
product = np.multiply(array,arr_2) #Element-wise multiplication
quotient = np.divide(array,arr_2) #Element-wise division
print("Element-wise addition of two arrays is : \n",add)
print("Element-wise subtraction of two arrays is : \n",difference)
print("Element-wise multiplication of two arrays is : \n",product)
print("Element-wise division of two arrays is : \n",quotient)
squares = np.power(array,2) #Element-wise squares of the first array
square_roots = np.sqrt(array) #Element-wise square-rooots of the first array
```

```
print("Element-wise squares of the first array : \n", squares)
print("Element-wise square-roots fo the first array \n: ",square_roots)
```

```
the sliced array is :
[[ 2  3  4]
 [ 7  8  9]
 [12 13 14]]
Reshaped array is :
[[ 1  2  3]
 [ 4  5  6]
 [ 7  8  9]
 [10 11 12]
 [13 14 15]]
the element at the position (1,3) in the array is :  9
Element-wise addition of two arrays is :
 [[12 14 16 18 20]
 [22 24 26 28 30]
 [32 34 36 38 40]]
Element-wise subtraction of two arrays is :
 [[-10 -10 -10 -10 -10]
 [-10 -10 -10 -10 -10]
 [-10 -10 -10 -10 -10]]
Element-wise multiplication of two arrays is :
 [[ 11  24  39  56  75]
 [ 96 119 144 171 200]
 [231 264 299 336 375]]
Element-wise division of two arrays is :
 [[0.09090909 0.16666667 0.23076923 0.28571429 0.33333333]
 [0.375      0.41176471 0.44444444 0.47368421 0.5       ]
 [0.52380952 0.54545455 0.56521739 0.58333333 0.6       ]]
Element-wise squares of the first array :
 [[  1   4   9  16  25]
 [ 36  49  64  81 100]
 [121 144 169 196 225]]
Element-wise square-roots fo the first array
:  [[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]]
```

DATA AGGREGATION

The basic aggregation operations using numpy are as follows:

Sum of All Elements:

The code calculates the total sum of all elements in the array using np.sum(array). This operation adds up all the values in the array and provides a single numerical result. Mean (Average) of All Elements:

The mean is computed using np.mean(array). The mean represents the average of all the elements

in the array, which is calculated by dividing the sum of all elements by the number of elements. Median Value:

The median is found using np.median(array). The median is the middle value when the array's elements are sorted in ascending order. If the array has an even number of elements, the median is the average of the two middle values. Variance of Elements:

Variance is computed using np.var(array). Variance measures how much the elements in the array vary from the mean. A higher variance indicates that the data points are spread out more widely from the mean. Standard Deviation:

The standard deviation is calculated using np.std(array). It is the square root of the variance and provides a measure of the spread of the data. Like variance, a higher standard deviation indicates greater variability in the data. Minimum Value:

The minimum value in the array is found using np.min(array). This operation returns the smallest value in the array. Maximum Value:

The maximum value in the array is determined using np.max(array). This operation returns the largest value in the array. After computing these aggregations, the code prints out the results, giving a comprehensive summary of the data in terms of sum, mean, median, variance, standard deviation, minimum, and maximum values.

Grouping Data After performing the aggregations, the code moves on to demonstrate data grouping based on a condition. Grouping is a way of categorizing data into different subsets based on specified criteria.

Group 1:

The code creates group_1, which contains all elements in the array that are greater than 7. This is done using the condition array[array $>$ 7]. The resulting group will only include elements that satisfy this condition. Group 2:

Similarly, group_2 contains all elements in the array that are less than or equal to 7. This group is created using the condition array[array $<=$ 7].

The sample code for the every tiopic mentioned above is provided in the below shell.

[34]:
```
#DATA AGGREGATION
sum = np.sum(array)        # Sum of all elements
mean = np.mean(array)       # Mean (average) of all elements
median = np.median(array)     # Median value
variance = np.var(array)        # Variance of elements
standard_deviation = np.std(array)        # Standard deviation
min_value = np.min(array)        # Minimum value
max_value = np.max(array)        # Maximum value
print("The sum of all elements in the array is  :" , sum)
print("The mean of the all elements in the array is : ",mean)
print("The median of the all elements in the array is : ",median)
print("The variance of the all elements in the array is :",variance)
print("The standard deviation of all elements in the array is :␣
 ↪",standard_deviation)
```

```
print("The minimum value in the array is : ",min_value)
print("The maximum value in the array is : ",max_value)
print("\n GROUPING DATA")
#grouping data based on condition
group_1 = array[array > 7]
group_2 = array[array<= 7]
print("group_1 elements (>7) : ", group_1)
print("group_2 elements (<=7) : ", group_2)
```

```
The sum of all elements in the array is  : 120
The mean of the all elements in the array is :  8.0
The median of the all elements in the array is :  8.0
The variance of the all elements in the array is : 18.666666666666668
The standard deviation of all elements in the array is :  4.320493798938574
The minimum value in the array is :  1
The maximum value in the array is :  15

 GROUPING DATA
group_1 elements (>7) :  [ 8  9 10 11 12 13 14 15]
group_2 elements (<=7) :  [1 2 3 4 5 6 7]
```

DATA ANALYSIS

Correlation

Correlation Coefficient Matrix: The code calculates the correlation between two predefined arrays (array and arr_2) using the np.corrcoef() function. Correlation measures the strength and direction of the linear relationship between two variables. The result is a correlation coefficient matrix.

The matrix is a 2x2 array where: The diagonal elements represent the correlation of each variable with itself (always 1). The off-diagonal elements represent the correlation between the two different variables. A correlation coefficient close to 1 indicates a strong positive linear relationship, -1 indicates a strong negative linear relationship, and 0 indicates no linear relationship.

Identifying Outliers

Outliers Detection: The code generates a random array of data using a normal distribution and then identifies outliers based on the Z-score. Z-score: The Z-score indicates how many standard deviations a data point is from the mean. It is calculated by subtracting the mean from the data point and dividing by the standard deviation.

Outliers: In this context, outliers are defined as data points with a Z-score greater than 3 (i.e., data points that are more than 3 standard deviations away from the mean). The code identifies these outliers by filtering the array for Z-scores greater than 3 and prints them. These outliers are considered unusual or extreme values compared to the rest of the data.

Calculating Percentiles

Percentiles Calculation: The code calculates specific percentiles (25th, 50th, and 75th) of the array. Percentiles: Percentiles are values below which a certain percentage of data falls. They help describe the distribution of data: 25th Percentile: Also known as the first quartile (Q1), it indicates that 25% of the data falls below this value. 50th Percentile: Also known as the median, it indicates

that 50% of the data falls below this value. It is a measure of central tendency. 75th Percentile: Also known as the third quartile (Q3), it indicates that 75% of the data falls below this value. The percentiles provide insight into the spread and distribution of the data, helping to understand the range and skewness of the dataset.

The sample code for the every tiopic mentioned above is provided in the below shell.

[51]:
```
#DATA ANALYSIS
#CORRELATION
correlation_matrix = np.corrcoef(array,arr_2)#correlation coefficient martix␣
 ↪for the predefined arrays
print("Correlation Coefficient Matrix:\n", correlation_matrix)

#IDENTIFYING OUTLIERS
#let us generate a random array
arr = np.random.randn(1000)
z_scores = np.abs((arr - np.mean(arr)) / np.std(arr))#calculates the z-scores␣
 ↪for z-test
outliers = arr[z_scores > 3] #outliers
print("\nOutliers are : ",outliers)

#CALCULATING PERCENTILES
# Calculate specific percentiles
percentile_25 = np.percentile(arr, 25)
percentile_50 = np.percentile(arr, 50)
percentile_75 = np.percentile(arr, 75)
print("\npercentiles are :")
print(percentile_25,"\n",percentile_50,"\n",percentile_75)
```

```
Correlation Coefficient Matrix:
 [[1. 1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1. 1.]]

Outliers are :  [-3.60275696  3.47171822]

percentiles are :
-0.6343356053094915
 0.06122200554875143
 0.7363477485440859
```

Advantages of Using NumPy Over Traditional Python Data Structures Performance and Efficiency:

Speed: NumPy is significantly faster than traditional Python data structures like lists and tuples for numerical computations. This is due to its implementation in C and its use of vectorized operations, which eliminate the need for explicit loops. Memory Efficiency: NumPy arrays are

stored in contiguous blocks of memory, unlike Python lists, which are pointers to objects. This allows for faster access and processing, particularly when working with large datasets. Broad Functionality:

Mathematical and Statistical Operations: NumPy provides a wide range of built-in functions for mathematical and statistical operations. These functions are optimized for performance, allowing for efficient computation on large arrays of data. Linear Algebra: NumPy includes powerful linear algebra tools, which are essential in areas like machine learning for operations like matrix multiplication, eigenvalue decomposition, and singular value decomposition. Integration with Other Libraries: NumPy seamlessly integrates with other scientific and data analysis libraries such as Pandas, SciPy, and TensorFlow, making it a cornerstone of the data science ecosystem. Scalability:

Handling Large Datasets: NumPy is designed to handle large datasets efficiently. Its operations are highly optimized, making it possible to perform complex calculations on large datasets without compromising on performance. Parallelism and Broadcasting: NumPy supports broadcasting, which allows operations on arrays of different shapes without requiring explicit replication of data. This feature is particularly useful in machine learning and image processing tasks. Real-World Examples Where NumPy's Capabilities Are Crucial Machine Learning:

In machine learning, NumPy is used for data preprocessing, model implementation, and evaluation. For example, the core computations in deep learning models, such as matrix multiplications and gradient calculations, are heavily reliant on NumPy. Example: Training a neural network involves multiple matrix operations, such as the multiplication of input data with weights, which NumPy handles efficiently. Financial Analysis:

NumPy is widely used in financial modeling, risk analysis, and algorithmic trading. It enables quick calculations of returns, volatilities, correlations, and other key financial metrics. Example: Calculating the Value at Risk (VaR) for a portfolio involves simulations and statistical analysis, where NumPy is used to perform large-scale computations efficiently. Scientific Research:

In scientific research, particularly in fields like physics, biology, and chemistry, NumPy is used for data analysis, simulation, and modeling. Researchers rely on NumPy for handling large datasets and performing complex numerical calculations. Example: In genomics, analyzing large datasets of genetic information to identify patterns and mutations involves extensive use of NumPy for efficient data processing and analysis.