PER SCHOLAS

Python NumPy

Introduction to lesson

- This lesson provides learners with more information about what NumPy is and what it does in Python. This lesson is designed for beginners.
- NumPy provides powerful tools for creating and modifying arrays efficiently, which is fundamental for numerical and scientific computing.

Learning Objectives

By the end of the lesson, learners will be able to:

- Describe the NumPy.
- Explain the Data Type Objects in NumPy.
- Define one-dimensional and multi-dimensional arrays in NumPy.
- Use Numpy in Python programming.
- Define how to create, manipulate, and work with multi-dimensional arrays.
- Use indexing to access a specific element in the array.
- Perform element-wise arithmetic operations with scalar values (1 and 2) using NumPy.
- Use NumPy aggregate functions for calculation.
- Use the NumPy broadcasting feature to multiply the entire array by scalar values.

Table of Contents

Overview of Array Data Structure in Python

- An array in Python is a compact way of collecting basic data types. All of the entries in an array must be of the same data type; however, arrays are not popular in Python, unlike other programming languages such as C# or Java.

- In general, when people talk of arrays in Python, they are actually referring to lists. There is a fundamental difference between them. In Python, arrays can be seen as a more efficient way of storing certain kinds of lists, which would contain elements of the same data type.

- In Python, there is no built-in array data structure; alternatively, we can use a list for that function. We can implement arrays using **array** modules or **NumPy** modules.

Introduction to Numpy

- NumPy stands for ‘Numerical Python.’ NumPy is a Python library for working with arrays and numerical data.

- NumPy provides a powerful **ndarray** object, which is a multi-dimensional array that can store homogeneous data types, such as integers, floats, or booleans.

- NumPy also provides a wide range of functions and methods for working with arrays, including mathematical operations, slicing and indexing, reshaping, and more. NumPy arrays are often used in scientific computing, data analysis, and machine learning.

Python numpy official website page provides more information about the various type codes available and the functionalities provided by the numpy module.

Reasons to use NumPy

- **Efficient Array Operations**: NumPy provides a multidimensional array object that is highly efficient for performing numerical operations. These arrays are contiguous blocks of memory, allowing for fast element-wise operations and vectorized computations.
- **Support for Multidimensional Arrays:** NumPy supports n-dimensional arrays, making it suitable for a wide range of applications, including data analysis, machine learning, scientific computing, and simulations.
- **Broadcasting:** NumPy allows you to perform operations on arrays of different shapes without explicitly writing loops. This feature, known as broadcasting, simplifies many common tasks, and enhances code readability.
- **Mathematical and Statistical Functions:** NumPy provides a vast collection of mathematical and statistical functions, including mean, median, standard deviation, correlation, and more. These functions are essential for data analysis and scientific computations.
- **Data Processing:** NumPy is commonly used for data preprocessing, cleaning, and transformation. It is an integral part of the data science and machine learning stack.
- **Integration with Other Libraries:** NumPy integrates well with other Python libraries used in scientific computing such as SciPy (for more advanced scientific functions), matplotlib (for data visualization), and scikit-learn (for machine learning).

How to Install NumPy Python

- Installing the NumPy library is a straightforward process. You can use pip to install the library.
- Go to the command line and type the following:

```
pip install numpy
```

Note:

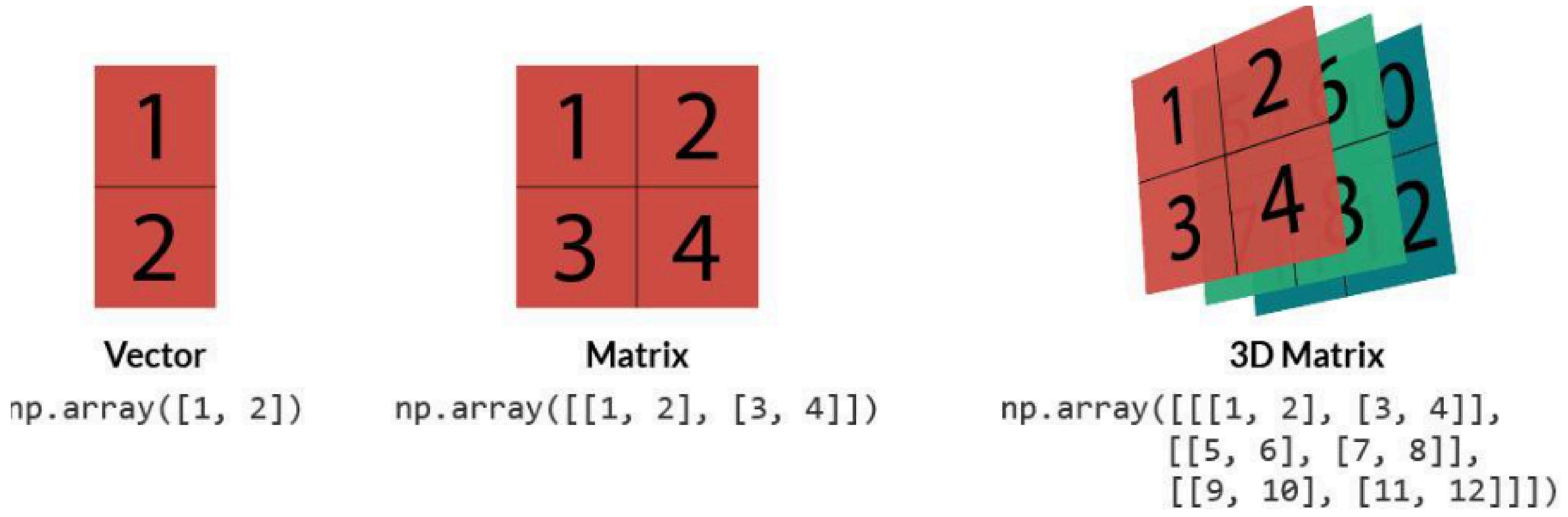
- If you are using Anaconda distribution, you can use conda to install NumPy.

```
conda install numpy
```

Reference: https://numpy.org/install/

Overview of NumPy Arrays

- **The NumPy array:** an n-dimensional data structure - is the central object of the NumPy package.
- A one-dimensional NumPy array can be thought of as a vector, a two-dimensional array as a matrix (i.e., a set of vectors), and a three-dimensional array as a tensor (i.e., a set of matrices).



```
np.array([1, 2])
```

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

```
np.array([[[1, 2], [3, 4]],
          [[5, 6], [7, 8]],
          [[9, 10], [11, 12]]])
```

Img src: learndatasci.com

Ways to Create NumPy Array

- NumPy arrays can be create or initialize using various functions such as **np.array(), np.zeros(), np.ones(), np.arange(), np.linspace(), np.eye(), np.random()**, etc.
- In the following example, we imported NumPy module as np and created a one-dimensional array with five elements using the np.array() function. The array is printed using the print() function.

Example 1: Creating a one-dimension arrays using np.arange()

```
import numpy as np
# Creating a one-dimensional array with 5
elements
a = np.array([1, 2, 3, 4, 5])
print(a)
```

NumPy - Ndarray Object

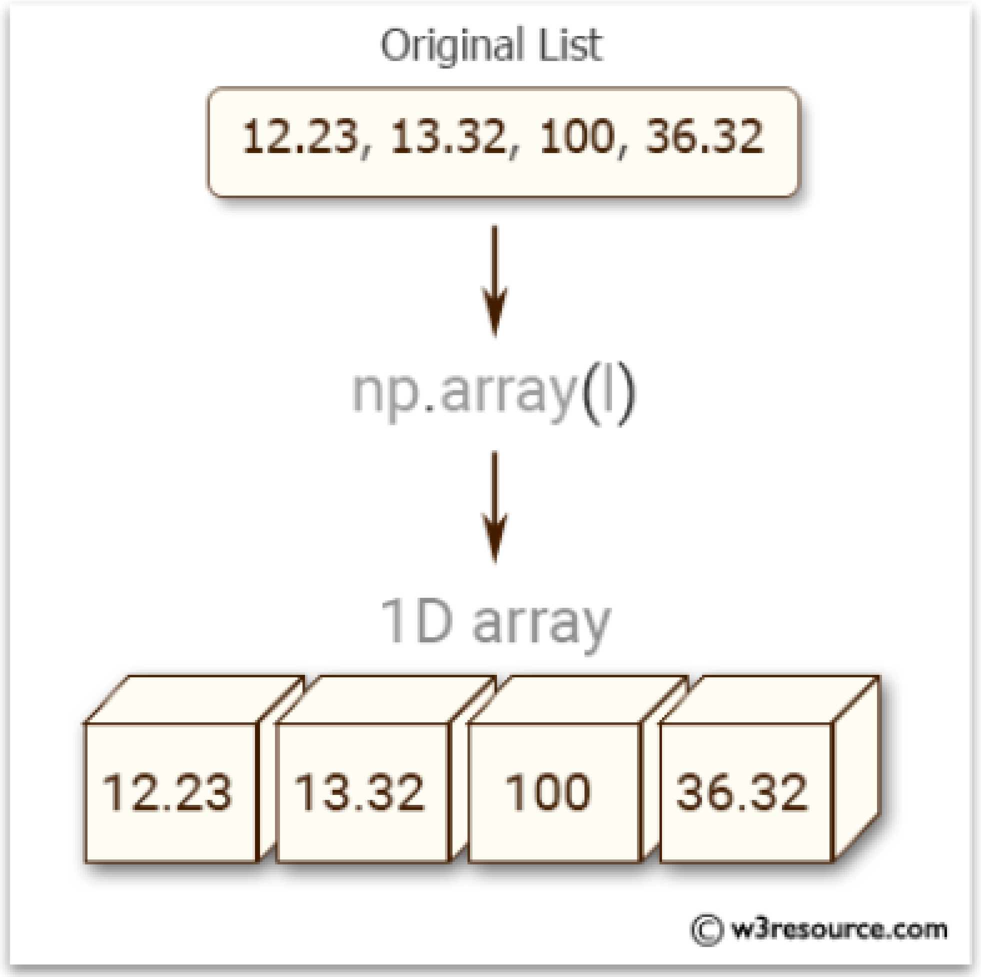
- The most important object defined in NumPy is an **N-dimensional** array type called **ndarray**. It describes the collection of items of the same type. Items in the collection can be accessed using a zero-based index.
- Every item in an **ndarray** takes the same size of block in the memory. Each element in ndarray is an object of data-type object (called dtype).
- The basic ndarray is created using an array function in NumPy as follows –

```
import numpy as np
np.array
#output <function numpy.array>

a = np.array([1,2,3])
print a
```

NumPy - Ndarray Object

- One-dimensional arrays are created using the **numpy.array()** function, which takes a sequence (e.g., a list, tuple, or range) as its argument.
- The data type of the array is determined automatically based on the data type of the elements in the input sequence. However, you can also explicitly specify the data type using the dtype argument.
- You can perform various operations on one-dimensional arrays such as slicing and indexing, mathematical operations, and boolean operations. For example, you can use array slicing to select a subset of the array such as my_array[1:4], or use mathematical operations to perform element-wise addition or multiplication with another array such as my_array + other_array.



Example: One-Dimensional Array Using array()

- In this example, we will create a one-dimensional NumPy array with 5 elements: 1, 2, 3, 4, and 5 by using **np.array()** function
- The **np.array()** function takes a list as its argument and returns a new **'ndarray'** object.

```
import numpy as np

# Creating a 1-dimensional NumPy array with 5 elements
my_array = np.array([1, 2, 3, 4, 5])
print(my_array)
```

Output:

```
[1 2 3 4 5]
```

Overview of arange() Function

- NumPy **arange()** is one of the array creation routines based on numerical ranges.
- It creates an instance of **ndarray** with evenly spaced values within a given interval and returns the reference to it.
- Values are generated within the half-open interval [start, stop); in other words, the interval including start, but excluding stop. For integer arguments, the function is equivalent to the Python built-in range() function, but returns an ndarray rather than a list.
- The format of the function is as follows –

numpy.arange (start, stop, step, dtype)

  - **Start:** The start of an interval. If omitted, defaults to 0.
  - **Stop:** The end of an interval (not including this number).
  - **Step:** Spacing(difference) between values, default is 1.
  - **dtype:** Data type of resulting ndarray. If not given, data type of input is used.

Example: One-Dimensional Array using arange()

Creating arrays using np.arange(): In this example, we will create a one-dimensional NumPy array by using **np.arange()** function.

Example 1: We create a vector with values spanning 1 up to 30 (but not including):

```
import numpy as np
MyArray = np.arange(1,30)
print(MyArray)
```

Example 2:

```
import numpy as np
# start and stop parameters set
x = np.arange(10,20,2)
print (x)
```

Example 3:

```
import numpy as np

x = np.arange(-3, 10,
4, dtype=int)
print (x)
```

NumPy Array Attributes

NumPy array has various attributes, including but not limited to:

- **ndim –** the number of dimensions of the array.
- **shape –** the shape of the array. This is a tuple of integers indicating the size of the array in each dimension.
- **size –** the total number of elements of the array
- **dtype –** an object describing the type of the elements in the array: standard Python types, int32, int16, float64, etc.

```
import numpy as np
my_array = np.array([[1, 2, 3, 4, 5], [69,58,74,32,9]])
# Accessing elements of the array using indexing
print(my_array.ndim)
print(my_array.shape)
print(my_array.size)
print(my_array.dtype)
```

Output:

```
2
(2, 5)
10
 int64
```

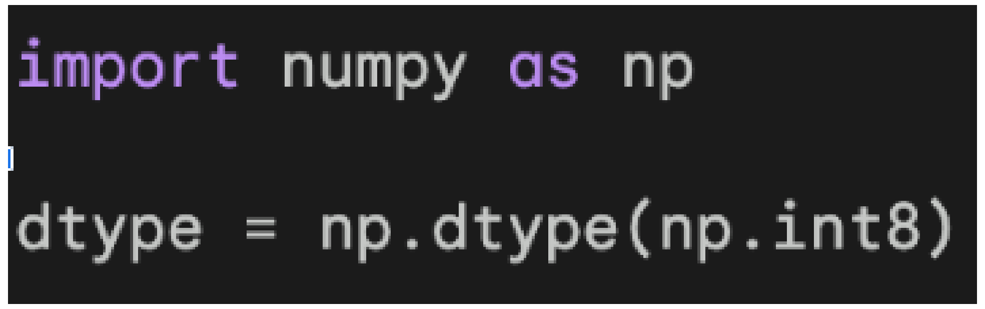
dtype - Data Type Objects in NumPy

- In NumPy, **dtype (data type)** refers to an object that specifies the type of data stored in a NumPy array. It defines how the data is stored in memory and how different operations should be carried out on that data. Each NumPy array has a **dtype** attribute that indicates the type of elements it contains.
- It describes the following aspects of the data:
  - Type of the data (integer, float, Python object, etc.).
  - Size of the data (number of bytes).
  - The byte order of the data (little-endian or big-endian).
  - If the data type is a sub-array, what is its shape and data type?

Note:

dtype is different from type.

- A data type object is an instance of the **NumPy.dtype** class and it can be created using **NumPy.dtype**. For example, the following code creates a data type object for an integer with 8 bits of precision.



```python
import numpy as np

dtype = np.dtype(np.int8)
```

Reference: https://numpy.org/doc/stable/reference/arrays.dtypes.html

Common NumPy Data Types

Common NumPy data types include:

- **int:** Integer types (e.g., int8, int16, int32, int64)
- **uint:** Unsigned integer types (e.g., uint8, uint16, uint32, uint64)
- **float:** Floating-point types (e.g., float16, float32, float64)
- **complex:** Complex types (e.g., complex64, complex128)
- **bool:** Boolean type (bool)
- **object:** Object type (generic Python object)

Reference: https://numpy.org/doc/stable/reference/arrays.dtypes.html

Example - Data Type Objects in NumPy

- The data type is crucial because it determines the size of each element in the array and how the elements are interpreted when performing mathematical operations. For example, a NumPy array with **dtype='int16'** will store each integer using 16 bits, and arithmetic operations will be carried out in a way consistent with 16-bit integer arithmetic.
- Example:

```
import numpy as np
# Creating arrays with different data types
arr_int32 = np.array([1, 2, 3], dtype=np.int32)
arr_float64 = np.array([1.0, 2.0, 3.0], dtype=np.float64)
arr_complex = np.array([1 + 2j, 3 + 4j],
dtype=np.complex128)

# Checking the dtype attribute
print(arr_int32.dtype)     # Output: int32
print(arr_float64.dtype)   # Output: float64
print(arr_complex.dtype)   # Output: complex128
```

Knowledge Check

How can you explicitly specify the data type of a one-dimensional NumPy array during creation?

- A) By using the numpy.dtype() function.
- B) By setting the data type directly within the array using my_array.dtype = 'desired_dtype'.
- C) By using the numpy.array() function with the dtype argument.
- D) By casting the array using the int() or float() functions.

Multidimensional Array

- In NumPy, a multidimensional array is represented by the **ndarray** object, which can have any number of dimensions (or axes).

- You can create a multidimensional array using the **numpy.array()** function and **passing a nested list or tuple as the argument**. For example,
  - **my_array = numpy.array([[1, 2, 3], [4, 5, 6]])** creates a two-dimensional array with two rows and three columns.

- Multidimensional arrays can be indexed and sliced in a similar way to one-dimensional arrays, but with multiple indices or slices. For example,
  - **my_array[1, 2]** selects the element in the second row and third column, and **my_array[:, 1:3]** selects all rows in the second and third columns.

Example - Multidimensional Array

The following code block creates a two-dimensional array with three rows and two columns.

```
import numpy as np
import numpy as np

# Creating a 2-dimensional NumPy array with 3 rows and 2 columns
my_array = np.array([[1, 2], [3, 4], [5, 6]])
print(my_array)
# Creating a 2-dimensional NumPy array with 3 rows and 2 columns
my_array = np.array([[1, 2], [3, 4], [5, 6]])
print(my_array)
```

Output

```
[  [1 2]
   [3 4]
  [5 6]  ]
```

Random Numbers in NumPy

NumPy offers the **random** module to work with random numbers. This module is useful for generating random data for the visualization. We can import **random** module by using below code.

```
from numpy import random
```

The **random** module provides several functions for generating random numbers. The most commonly used functions are:

- np.random.**rand():** Generates random numbers uniformly distributed between 0 and 1.
- np.random.**randn()**: Generates random numbers from a standard normal distribution with mean 0 and standard deviation 1.
- np.random.**randint()**: Generates random integers between a specified range.
- np.random.**choice()**: Generates random samples from a given sequence.
- np.random.**shuffle()**: Shuffles a sequence in place.

Click here for the examples - Guided Lab - 343.2.1 - NumPy and Mathematical Calculation

NumPy Operations

NumPy is a popular Python library used for numerical computing. It provides an efficient and convenient interface for performing numerical operations on arrays and matrices. Here are some of the most common operations that can be performed using NumPy:

- Indexing and slicing
- Arithmetic operations
- Aggregate functions
- Broadcasting
- Reshaping and Transposing
- Linear Algebra operations
- Boolean operations
- Sorting
- Masking

Example 1 - Indexing and Slicing

In this example, we create a one-dimensional NumPy array named arr with five elements. We then use indexing to access a specific element in the array (arr[2] returns the element at index 2, which is 3). We also use slicing to access a range of elements in the array (arr[1:4] returns the elements at indices 1, 2, and 3).

```
import numpy as np
# Create a 1D numpy array
arr = np.array([1, 2, 3, 4, 5])
# Indexing: Access a specific element in the array
element = arr[2] # Returns the element at index 2 (which is 3)
print(element)
# Slicing: Access a range of elements in the array
slice = arr[1:4] # Returns elements at indices 1, 2, and 3
print(slice)
```

Output

```
3
[2 3 4]
```

Example 2 - Indexing and Slicing

You can access and modify the elements of the array using indexing and slicing like this:

```
import numpy as np

# Creating a 1-dimensional NumPy
array with 5 elements
my_array = np.array([1, 2, 3, 4, 5])

# Accessing elements of the array
using indexing
print(my_array[0])  # Output: 1
print(my_array[3])  # Output: 4
```

```
# Modifying elements of the array using indexing
my_array[2] = 7
print(my_array) # Output: [1 2 7 4 5]

# Accessing a subset of the array using slicing
print(my_array[1:4]) # Output: [2 7 4]

# Modifying a subset of the array using slicing
my_array[1:4] = np.array([6, 8, 9])
print(my_array)

# Output: [1 6 8 9 5]
```

Python NumPy - Arithmetic Operations

In this example, we created a one-dimensional NumPy array(arr) with three elements. We then performed element-wise arithmetic operations with scalar values (1 and 2) using NumPy.

```
import numpy as np
# Create a 1D numpy array
arr = np.array([1, 2, 3])
# Perform arithmetic operations
sum_arr = arr + 1  # Element-wise sum with scalar 1
prod_arr = arr * 2 # Element-wise product with scalar 2
div_arr = arr / 2  # Element-wise division with scalar 2
# Print the results
print(sum_arr)
print(prod_arr)
print(div_arr)
```

Output

```
[2 3 4]
[2 4 6]
[0.5 1.  1.5]
```

Python NumPy - Aggregate functions

In this example, we use NumPy's aggregate functions to calculate the sum of all elements in arrary (arr_sum), the mean of all elements in arr (arr_mean), the minimum value in arr (arr_min), and the maximum value in arr (arr_max).

```
import numpy as np

# Create a 1D numpy array
arr = np.array([1, 2, 3, 4, 5])
# Perform aggregate functions
arr_sum = np.sum(arr) # Sum of all elements
arr_mean = np.mean(arr) # Mean of all elements
arr_min = np.min(arr) # Minimum value
arr_max = np.max(arr) # Maximum value

# Print the results
print(arr_sum, arr_mean, arr_min, arr_max, sep=',' )
```

Output

```
15, 3.0, 1, 5
```

Click here to see all available functions.

Knowledge Check

In a one-dimensional NumPy array, if you use slicing with the expression `**arr[1:4]**`, which elements will be returned?

A) The elements at indices 1 and 4.

B) The elements at indices 1, 2, and 3.

C) The elements at indices 2 and 4.

D) The elements at indices 1, 4, and 7.

Broadcasting

In this example, we will create a 2D array named ‘arr’ with dimensions 2x3, then we will use Numpy's broadcasting feature to multiply the entire array by a scalar value of 2. This operation multiplies each element in arr by 2, effectively doubling the values in the array.

```
import numpy as np
# Create a 2D numpy array

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

# Multiply the array by a scalar value
result = arr * 2

# Print the result
print(result)
```

Output

```
[[2  4  6]
 [8 10 12]]
```

Reshaping and Transposing

In the following example, we create a 2D numpy array matrix with 2 rows and 3 columns. We then use numpy's **T** attribute to transpose the matrix, effectively swapping its rows and columns. We store the transposed matrix in a variable called transposed_matrix. We then use numpy's reshape function to reshape the transposed matrix into a 2D matrix with 3 rows and 2 columns. We store the reshaped matrix in a variable called reshaped_matrix.

```
import numpy as np
# Create a 2D numpy array
matrix = np.array([[1, 2, 3], [4, 5, 6]])
# Transpose the matrix using numpy's attribute
transposed_matrix = matrix.T
# Reshape the transposed matrix into a 3x2 matrix
reshaped_matrix = transposed_matrix.reshape(3, 2)
# Print the original matrix, transposed matrix, and reshaped matrix
print("Original Matrix:")
print(matrix)
print(transposed_matrix)
print(reshaped_matrix)
```

Output

```
Original Matrix:
[[1 2 3]
 [4 5 6]]

Transposed Matrix:
[[1 4]
 [2 5]
 [3 6]]

Reshaped Matrix:
[[1 4]
 [2 5]
 [3 6]]
```

Linear Algebra Operations

Here are some examples of common linear algebra operations using Python's NumPy library:

1. Vector (one dimension) Addition
2. Matrix (two dimension) Addition
3. Vector (one dimension) Dot Product
4. Matrix (two dimension) Multiplication
5. Transpose of a Matrix

Vector (one dimension) Addition

In this example, we perform vector addition using numpy arrays. Vector addition involves adding corresponding elements of two vectors (arrays) together to create a new vector.
We start by creating two numpy arrays, ‘a’ and ‘b’. To add the vectors ‘a’ and ‘b’, we simply use the **‘+’** operator between them. Numpy handles the element-wise addition.

Finally, we print the result using the print function.

```
import numpy as np
a = np.array([1, 2, 3])
b = np.array([4, 5, 6])
c = a + b

print(c) #[5 7 9]
```

Matrix (two dimension) Addition

In this example, we perform matrix addition using NumPy arrays. Matrix addition involves adding corresponding elements of two matrices together to create a new matrix. We start by creating two numpy matrices, ‘A’ and ‘B.’ To add the matrices ‘A’ and ‘B,’ we simply use the **‘+’** operator between them. Numpy handles the element-wise addition.

Finally, we print the result using the print function:

```
import numpy as np

A = np.array([[1, 2], [3, 4]])
B = np.array([[5, 6], [7, 8]])
C = A + B

print(C)
```

Output:

```
[[ 6 8]
 [10 12]]
```

Vector (one dimension) Dot Product

In this example, we calculate the dot product of two NumPy arrays, ‘a‘ and ‘b,’ using the np.dot() function. We start by creating two numpy arrays, ‘a’ and ‘b.’ The dot product of two arrays is calculated using the **np.dot()** function. The dot product of a and b is the sum of the products of their corresponding elements.

Finally, we print the result using the print function.

```
import numpy as np
a = np.array([1, 2, 3])
b = np.array([4, 5, 6])
c = np.dot(a, b)
print(c)
```

Output:

```
32
```

Matrix (two dimension) Multiplication

In this example, we perform matrix multiplication using numpy arrays. Matrix multiplication involves multiplying corresponding elements of two matrices and summing the results to obtain the new matrix. We start by creating two numpy matrices, ‘A’ and ‘B.’ To multiply the matrices ‘A’ and ‘B’, we use the **np.dot()** function.

Finally, we print the result using the print function.

```
import numpy as np

A = np.array([[1, 2], [3, 4]])
B = np.array([[5, 6], [7, 8]])
C = np.dot(A, B)

print(C)
```

Output:

```
[ [19 22]
 [43 50] ]
```

Transpose of a Matrix

In this example, we perform matrix transposition using the **np.transpose()** function. Transposing a matrix involves swapping its rows and columns. We start by creating a numpy matrix ‘A.’ To transpose the matrix, we use the **np.transpose()** function. Finally, we print the result using the print function.

```
import numpy as np

A = np.array([[1, 2], [3, 4]])
B = np.transpose(A)
print(B)
```

Output

```
[[1 3]
[2 4]]
```

NumPy - Boolean Operations

Here are some Boolean operations we can utilize with Python's NumPy library:

1. Element-Wise comparison
2. Logical AND
3. Logical OR
4. Logical NOT
5. Array-wise comparison

Element-Wise Comparison

In this example, we perform element-wise comparison of two numpy arrays, ‘a’ and ‘b’ using the == operator. Element-wise comparison involves comparing each element of one array with the corresponding element of the other array.

```
import numpy as np
a = np.array([1, 2, 3, 4, 5])
b = np.array([2, 2, 3, 3, 5])
c = a == b
print(c)
```

Output

```
[False  True  True False  True]
```

Logical AND

In this example, we perform a logical AND operation between two numpy boolean arrays, ‘a’ and ‘b’ using the **np.logical_and()** function. Logical AND involves determining the truth value of the conjunction of corresponding elements from both arrays.

```
import numpy as np

a = np.array([True, True, False, False])
b = np.array([True, False, True, False])
c = np.logical_and(a, b)
print(c) # Output: [ True False False False]
```

Output

```
[ True False False False]
```

Logical OR

In this example, we perform a logical OR operation between two NumPy Boolean arrays, ‘a’ and ‘b’ using the **np.logical_or()** function. Logical OR involves determining the true value of the disjunction of corresponding elements from both arrays.

```
import numpy as np

a = np.array([True, True, False, False])
b = np.array([True, False, True, False])
c = np.logical_or(a, b)

print(c)
```

Output

```
[ True True True False]
```

Logical NOT

In this example, we perform a logical NOT operation on a numpy boolean array ‘a’ using the **np.logical_not()** function. Logical NOT involves negating the true value of each element in the array.

```
import numpy as np

a = np.array([True, False, True, False])
b = np.logical_not(a)

print(b)
```

Output

```
[False True False True]
```

Logical NOT

In this example, we perform a logical NOT operation on a numpy boolean array ‘a’ using the np.logical_not() function. Logical NOT involves negating the true value of each element in the array.

```
import numpy as np

a = np.array([True, False, True, False])
b = np.logical_not(a)

print(b)
```

Output

```
[False True False True]
```

Array-Wise Comparison

In this example, we perform an array-wise comparison between two numpy arrays, ‘a’ and ‘b’, using the **np.array_equal()** function. This function checks if the two arrays are element-wise equal.

```
import numpy as np

a = np.array([1, 2, 3, 4, 5])
b = np.array([2, 2, 3, 3, 5])
c = np.array_equal(a, b)

print(c)
```

Output

```
False
```

Knowledge Check

What does an element-wise comparison of two NumPy arrays involve?

A) Checking if both arrays have the same number of elements.

B) Comparing each element of one array with the corresponding element of the other array.

C) Checking if the arrays have identical data types.

D) Concatenating the two arrays and comparing the resulting concatenated array.

Sorting

In this example, we first create a NumPy array (a) with some random values. We then use the **np.sort()** function to sort the array in ascending order and store the sorted array in a new variable b. Finally, we print the sorted array (b).

```
import numpy as np

a = np.array([3, 1, 4, 1, 5, 9, 2, 6, 5, 3])
b = np.sort(a)

print(b)
```

Output

```
[1 1 2 3 3 4 5 5 6 9]
```

Masking

In this example, we first create a NumPy array named a, with some random values. We then create a boolean mask by comparing each element in the array (a) with the value 3 (elements greater than 3). The result is a boolean array with True values, where the condition is satisfied and False values, where it is not. We then use this mask to select only the elements of (a) where the mask is True, and store them in a new variable (b). Finally, we print the masked array (b).

```
import numpy as np
a = np.array([3, 1, 4, 1, 5, 9, 2, 6, 5, 3])
mask = a > 3
b = a[mask]
print(b)
```

Output

```
[4 5 9 6 5]
```

Searching and Counting

The in operator can be used to check if a value is present in an array. The argmin() and argmax() functions can be used to find the index of the minimum and maximum values in an array, respectively.

```
import numpy as np
arr = np.array([33, 2, 3,56,58,96,4,6,9,])
x = 3
print("Is 3 in the array?")
print(3 in arr)
print("The index of the minimum value:")
print(np.argmin(arr))
print("The index of the maximum value:")
print(np.argmax(arr))
```

Counting

The **count_nonzero()** function can be used to count the number of non-zero elements in an array.

```
import numpy as np
# Example 1: Count non-zero elements in a 1D array
arr_1d = np.array([0, 5, 0, 8, 0, 3, 7, 0])
nonzero_count_1d = np.count_nonzero(arr_1d)

print("1D Array:", arr_1d)
print("Number of Non-Zero Elements:", nonzero_count_1d)

# Example 2: Count non-zero elements in a 2D array
arr_2d = np.array([[0, 2, 0, 4],
          [5, 0, 0, 8],
          [0, 0, 3, 0]])
nonzero_count_2d = np.count_nonzero(arr_2d)
print("2D Array:")
print(arr_2d)
print("Number of Non-Zero Elements:", nonzero_count_2d)
```

Output:

```
1D Array: [0 5 0 8 0 3 7 0]
Number of Non-Zero Elements: 4

2D Array:
[[0 2 0 4]
 [5 0 0 8]
 [0 0 3 0]]

Number of Non-Zero Elements: 5
```

Find Unique Element or Value in 1D Array

The **unique()** function in NumPy is used to find the unique elements of an array and returns them in sorted order. Here is an example:

```
import numpy as np
arr_1d = np.array([3, 1, 2, 3, 4, 2, 5, 1, 6, 4])
unique_elements_1d = np.unique(arr_1d)

print("1D Array:", arr_1d)
print("Unique Elements:", unique_elements_1d)

# Example 2: Find unique elements in a 2D array
arr_2d = np.array([[1, 2, 3],
          [4, 2, 5],
          [1, 6, 4]])
unique_elements_2d = np.unique(arr_2d)
print("2D Array:")
print(arr_2d)
print("Unique Elements:", unique_elements_2d)
```

Output:

```
1D Array: [3 1 2 3 4 2 5 1 6 4]
Unique Elements: [1 2 3 4 5 6]
2D Array:
[[1 2 3]
[4 2 5]
[1 6 4]]

Unique Elements: [1 2 3 4 5 6]
```

NumPy Array Manipulation Functions

- The Array manipulation functions of NumPy module helps us to perform changes in the array elements.
- Have a look at the below functions–
  - **numpy.reshape():** This function allows us to change the dimensions of the array without hampering the array values.
  - **numpy.concatenate():** Joins two arrays of the same shapes either in a row-wise or a column-wise manner.

```
import numpy as np
arr1 = np.arange(4)
print('Elements of an array1:\n',arr1)
arr2 = np.arange(4,8)
print('Elements of an array2:\n',arr2)
res1 = arr1.reshape(2,2)
print('Reshaped array with 2x2 dimensions:\n',res1)
res2 = arr2.reshape(2,2)
print('Reshaped array with 2x2 dimensions:\n',res2)
print("Concatenation two arrays:\n")
concat = np.concatenate((arr1,arr2),axis=0)
print(concat)
```

Output:

```
Elements of an array1:
 [0 1 2 3]
Elements of an array2:
 [4 5 6 7]
Reshaped array with 2x2 dimensions:
 [[0 1]
 [2 3]]
Reshaped array with 2x2 dimensions:
 [[4 5]
 [6 7]]
Concatenation two arrays:
 [0 1 2 3 4 5 6 7]
```

Numpy[] vs. List[]

- NumPy is typically better than Python lists when you are dealing with numerical or scientific computing tasks, especially when performance and efficiency are important. If you need to work with large datasets, perform complex mathematical or statistical operations, or require support for multidimensional arrays, NumPy is a more suitable choice.

- However, for general-purpose data storage and manipulation tasks that do not involve heavy numerical computation, Python lists can be more convenient and straightforward to use. The choice between NumPy and Python lists should be based on the specific requirements of your project.

- All elements in a NumPy array must have the same data type, which allows for efficient storage and vectorized operations; on the other hand, Lists can contain elements of different data types within the same list.

Guided Lab - NumPy

- Please follow the link below to the guided lab for Numpy.
    - Guided Lab - 343.2.2 - NumPy and Mathematical Calculation
- You can also find this lab on Canvas under the Assignments section.
- If you have questions while performing the lab activity, ask your instructors for assistance.

Practice Assignment

- Complete this assignment **"PA 343.2.1 - Practice Assignment - NumPy"** for practice. You will find the assignment on Canvas, under the Assignment section.

- Note: Use your office hours to complete this assignment. If you have technical questions while performing the practice assignment, ask your instructors for assistance.

Knowledge Check

- What is NumPy, and how is it used in scientific computing and data analysis?
- What are some benefits of using NumPy in terms of speed and memory usage?
- What is the difference between a one-dimensional and a multidimensional NumPy array, and how are they represented in memory?
- Which numpy function is used to calculate the dot product of two numpy arrays?

Summary

- NumPy is a Python library used for scientific computing and data analysis that provides fast and efficient numerical operations.
- NumPy arrays can be used to store and manipulate large arrays of data, with built-in functions for mathematical operations.
- One-dimensional arrays in NumPy are commonly used for representing time series data, while multidimensional arrays are used for image and signal processing.
- NumPy provides significant speed and memory benefits over built-in Python data structures like Lists due to its optimized algorithms and memory management.

References

https://numpy.org/doc/stable/user/basics.indexing.html

PER SCHOLAS

Questions?

PER SCHOLAS

UNLOCKING POTENTIAL

CHANGING THE FACE OF TECH