PER SCHOLAS

Numpy

PER SCHOLAS

Section One

Numpy

Learning Objectives:

This lesson provides learners with more information about what Numpy is and what it does in Python. This presentation is designed for beginners. By the end of this lesson, learners will be able to:

- Define NumPy and its uses in scientific computing and data analysis.
- Explain how NumPy is optimized for speed and memory usage in scientific computing and data analysis.
- Define one-dimensional and multi-dimensional arrays in NumPy.
- Use Numpy in python programming.

PER SCHOLAS

Topics

1. Introduction to Numpy.
2. Speed and Memory Benefits of Numpy.
3. One-Dimensional and Multidimensional Array.
4. Dataframe in relation to Numpy Matrix.
5. Numpy Operations.

PER SCHOLAS

Introduction to Numpy

- NumPy is a Python library for working with arrays and numerical data. It is short for "Numerical Python."

- 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.

PER SCHOLAS

Speed and Memory Benefits of Numpy

- NumPy provides significant speed and memory benefits over standard Python lists when working with numerical data.

- NumPy's *ndarray* object is implemented in C, which allows for fast computation and efficient memory usage compared to Python lists, which are implemented in Python itself.

- NumPy's ability to perform element-wise operations and broadcasting allows for very efficient computation of large arrays without using loops. This can result in significant speed improvements over Python's built-in functions and loops.

PER SCHOLAS

One-Dimensional Arrays

- 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.

PER SCHOLAS

One-Dimensional Array - Example

In this example, we created a one-dimensional NumPy array with 5 elements: 1, 2, 3, 4, and 5. 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]
```

PER SCHOLAS

One-Dimensional Array - Example

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]
```

PER SCHOLAS

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.

PER SCHOLAS

Multidimensional Array Example

This creates a two-dimensional array with three rows and two columns.

```
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)
```

Output

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

PER SCHOLAS

Multidimensional Array Example

You can access the elements of the array using indexing and slicing with multiple indices or slices, like this:

```
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]])


# Accessing elements of the array using
indexing with multiple indices

print(my_array[0, 1])    # Output: 2

print(my_array[2, 0])    # Output: 5
```

```
# Modifying elements of the array using
indexing with multiple indices

my_array[1, 1] = 9

print(my_array)          # Output: [[1
2]
                         #          [3
9]
                         #          [5
6]]


# Accessing a subset of the array using
slicing with multiple slices

print(my_array[1:3, 0:2]) # Output: [[3
9]
```

PER SCHOLAS

Multidimensional Array Example

You can also perform various operations on the array such as element-wise arithmetic, matrix multiplication, and more. For example:

```
import numpy as np
# Creating two 2-dimensional NumPy arrays
a = np.array([[1, 2], [3, 4], [5, 6]])
b = np.array([[7, 8], [9, 10], [11, 12]])
# Performing element-wise arithmetic on the arrays
c = a + b
print(c)
# Performing matrix multiplication on the arrays
d = np.dot(a, b.T)
print(d)
```

Output

```
[[ 8 10]
 [12 14]
 [16 18]]
[[ 23  29  35]
 [ 53  67  81]
 [ 83 105 127]]
```

Dataframe in Relation to Numpy Matrix

- A DataFrame can be thought of as a collection of NumPy matrices, where each matrix represents a column of the DataFrame.

- The rows and columns of a DataFrame are labeled, which allows for easy indexing and selection of subsets of the data.

- While a NumPy matrix can only contain elements of a single data type, a DataFrame can contain elements of different data types, such as integers, floats, and strings.

PER SCHOLAS

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:

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

PER SCHOLAS

Creating arrays

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

```
import numpy as np


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

Output

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

PER SCHOLAS

Indexing and Slicing

In this example, we create a 1D numpy array 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

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

PER SCHOLAS

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]
```

PER SCHOLAS

Aggregate functions

In this example, we use Numpy's aggregate functions to calculate the sum of all elements in arr (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
```

PER SCHOLAS

Broadcasting

In this example, we create a 2D numpy array arr with dimensions 2x3. We then 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]]
```

PER SCHOLAS

Reshaping and Transposing

In this 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
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]]
```

Linear Algebra Operations

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

a. Vector Addition
b. Matrix Addition
c. Vector Dot Product
d. Matrix Multiplication
e. Transpose of a Matrix

PER SCHOLAS

Victor Addition

```
import numpy as np

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

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

Output

```
[5 7 9]
```

PER SCHOLAS

Matrix Addition

```
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]]
```

Output

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

PER SCHOLAS

Vector Dot Product

```
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
```

Output

32

PER SCHOLAS

Matrix Multiplication

```
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]]
```

Output

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

PER SCHOLAS

Transpose of a Matrix

```
import numpy as np

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

print(B) # Output: [[1 3]
         #          [2 4]]
```

Output

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

PER SCHOLAS

Boolean Operations

Here are some examples of Boolean operations using Python's NumPy library:

a. Element-Wise comparison
b. Logical AND
c. Logical OR
d. Logical NOT
e. Array-wise comparison

PER SCHOLAS

Element-Wise Comparison

```
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]
```

Output

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

PER SCHOLAS

Logical AND

```
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]
```

PER SCHOLAS

Logical OR

```
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]
```

Output

[True True True False]

PER SCHOLAS

Logical NOT

```
import numpy as np

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

print(b) # Output: [False  True False
True]
```

Output

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

PER SCHOLAS

Array-Wise Comparison

```
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
```

Output

```
False
```

PER SCHOLAS

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]
```

Output

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

PER SCHOLAS

Masking

In this example, we first create a NumPy array (a) with some random values. We then create a boolean mask by comparing each element in the array (a) with the value 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]
```

Output

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

PER SCHOLAS

Hands-On Activities

Open following link on **Google colaboratory.**

ACT 341.2.2 - Python Practice Problems-General.ipynb

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?
- ➢ What are some advantages and disadvantages of using DataFrames over NumPy arrays for data analysis tasks?

PER SCHOLAS

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.
- ➢ DataFrames provide a convenient way to represent and manipulate data, with built-in methods for filtering, aggregating, and transforming data, making them a popular choice for data analysis tasks.

PER SCHOLAS

Questions