Week:7

Numpy:

NumPy is a general-purpose array-processing Python library which provides handy methods/functions for working **n-dimensional arrays**. NumPy is a short form for "**Numerical Python**". It provides various computing tools such as comprehensive mathematical functions, and linear algebra routines.

- NumPy provides both the **flexibility of Python** and the speed of **well-optimized** compiled C code.
- Its **easy-to-use syntax** makes it highly accessible and productive for programmers from any background.

It provides support for arrays and matrices, along with a collection of mathematical functions that can be performed on these arrays.

Creating arrays:

To create an array in NumPy, you can use the np.array() function. This function takes a list as an argument and returns an array. For example, to create a one-dimensional array of integers, you can use the following code:

import numpy as np

Create an alias with the **as** keyword while importing:

Now the NumPy package can be referred to as **np** instead of **numpy**.

NumPy is usually imported under the **np** alias.

Certainly! In NumPy, arrays can be created with various dimensions, also known as ranks. Let's create and examine arrays of different dimensions.

0 Dimension array

import numpy as np

```
a = np.array(43)
print(a)
```

output

```
43
```

1Dimension array:

```
Import as numpy as np
arr = np.array([1,2,3,4,5,6])
print(arr)
```

output:

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

Creating a 2-D array

A two-dimensional (2-D) array can be thought of as a matrix or a list of lists.

```
import numpy as np
arr_2d=np.array([[1,2,3],[4,5,6]])
print(arr_2d)
```

output

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

Ex:

```
import numpy as np
```

```
x = np.array(42)
y = np.array([1, 2, 3, 4, 5])
z = np.array([[1, 2, 3], [4, 5, 6]])
v = np.array([[[1, 2, 3], [4, 5, 6]], [[1, 2, 3], [4, 5, 6]]])

print(x)
print(y)
print(z)
print(v)
```

output:

```
42

[1 2 3 4 5]

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

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

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

Basic Data Types in NumPy Arrays

NumPy arrays are foundational structures in numerical computing, providing efficient storage and manipulation of homogeneous data. In NumPy, each element within an array shares the same data type, ensuring consistency and enabling optimized operations.

NumPy provides a rich set of data types to handle various numerical and non-numerical data efficiently. Here's a quick reference:

- **Integers**: np.int8, np.int16, np.int32, np.int64

  Represent signed integers with 8, 16, 32, or 64 bits respectively. Alternatively, shorthand notations like int1, int2, int4, int8 may be used.

  Unsigned Integers: np.uint8, np.uint16, np.uint32, np.uint64

Denote unsigned integers with 8, 16, 32, or 64 bits respectively.

- **Floating-Point**: np.float16, np.float32, np.float64

  Define floating-point numbers with 16, 32, or 64 bits of precision.

- **Complex**: np.complex64, np.complex128

Enable the representation of complex numbers, comprising real and imaginary components. complex64 uses 64 bits, while complex128 uses 128 bits for higher precision.

- **Boolean**: np.bool

  Represented by ?, boolean data types store either True or False values, internally stored as a byte.

- **String**: np.string_, np.unicode

  Used for storing sequences of characters, typically ASCII encoded_

- **Object**: np.object_
- **Structured**: Custom defined with np.dtype

By choosing the appropriate data type, you can optimize memory usage and computation speed, which is crucial for large datasets and intensive numerical computations.

Datatypes and Memory Storage in NumPy Arrays

Example 1 illustrates creating an array with the default storage for integers, resulting in a 32-bit integer array.

```
Import numpy  as np

arr1 = np.array([1, 2, 3, 4, 5], dtype=int)

print("Example #1:")

print("Array:", arr1)

print("Data type:", arr1.dtype)

print()
```

output:

```
Example #1:

Array: [1 2 3 4 5]

Data type: int32
```

Ex:2

demonstrates creating an array with a defined storage size for integers, specifying np.int16, which results in a 16-bit integer array.

```
arr2 = np.array([1, 2, 3, 4, 5], dtype=np.int16)

print("Example #2:")

print("Array:", arr2)

print("Data type:", arr2.dtype)

print()
```

output:

```
Example #2:

Array: [1 2 3 4 5]

Data type: int16
```

Ex:3

shows another way to achieve the same result as Example 2 by using a character code ('i2') with the desired storage value (16-bit) to specify the data type

```
import numpy as np

arr3 = np.array([1, 2, 3, 4, 5], dtype='i2')

print("Example #3:")

print("Array:", arr3)
print("Data type:", arr3.dtype)
```

```
output:

Example #3:

Array: [1 2 3 4 5]

Data type: int16
```

Ex:

```
import numpy as np

arr = np.array([1, 2, 3, 4], dtype='i4')

print(arr)
print(arr.dtype)
```

output:

```
[1 2 3 4]
int32
```

creating numpy array using the list

1. **sequence:** It is the python sequence which is to be converted into the py thon array.
2. **dtype:** It is the data type of each item of the array.
3. **order:** It can be set to C or F. The default is C.

Ex:1

```
import numpy as np
x=[12,32,43,2,2,12,3,5]
a=np.array(x)
print(type(a))
print(a)s  5]
```

output:

```
<class 'numpy.ndarray'>
[12,32,43,2,12,3,5]
```

Ex:2

numpy array using more than one list

```
import numpy as np
x=[[1,2,3,4,5],[6,7,8,9]]
a=np.asarray(x)
print(type(a))
print(a)
```

output:

```
<class 'numpy.ndarray'>
[list([1,2,3,4,5]) list([6,7,8,9])
```

Slicing arrays

We pass slice instead of index like this: [*start*:*end*].
We can also define the step, like this: [*start*:*end*:*step*].
If we don't pass start its considered 0
If we don't pass end its considered length of array in that dimension
If we don't pass step its considered 1

Example:

```
import numpy as np

arr = np.array([1, 2, 3, 4, 5, 6, 7,9,10])

print(arr[2:6])
```

output:

```
[3 4 5 6]
```

Ex 2:

```
import numpy as np

arr = np.array([1, 2, 3, 4, 5, 6, 7,9,10])

print(arr[2:])
```

output:

```
[ 3  4  5  6  7  9 10]
```

Ex:3

```
import numpy as np

arr = np.array([1, 2, 3, 4, 5, 6, 7,9,10])

print(arr[:3])
```

output:

```
[1 2 3]
```

Negative Slicing

Ex:1

```
import numpy as np

arr = np.array([1, 2, 3, 4, 5, 6, 7])

print(arr[-4:-1])
```

output:

```
[4,5,6]
```

Slicing 2-D Arrays

```
import numpy as np

arr = np.array([[1, 2, 3, 4, 5], [6, 7, 8, 9, 10]])

print(arr[1, 1:4])
```

output:
[7,8,9]

Indexing:

The indexes in NumPy arrays start with 0, meaning that the first element has i ndex 0, and the second has index 1 etc.

Ex:1
Get the first element from the following array

import numpy as np

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

print(arr[0])

output:
[1]

Ex:2

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

print(arr[3])

output:
[4]

Access 2-D Arrays

To access elements from 2-D arrays we can use comma separated integers repr esenting the dimension and the index of the element.

Think of 2-D arrays like a table with rows and columns, where the dimension r epresents the row and the index represents the column.

Ex:1
Access the element on the first row, second column:

```
import numpy as np
arr = np.array([[1,2,3,4,5], [6,7,8,9,10]])

print('2nd element on 1st row: ', arr[0, 1])
```

output:
2nd element on 1st row: 2

Ex:2

```
import numpy as np

arr = np.array([[1,2,3,4,5], [6,7,8,9,10]])

print('5th element on 2nd row: ', arr[1, 4])
```

output:
```
5th element on 2nd dim:  10
```

Ex:3

Get third and fourth elements from the following array and add them

```
import numpy as np

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

print(arr[2] + arr[3])
```

output:
```
[7]
```

Numpy.arange

it creates an array by using the evenly spaced values over the given interval.
import numpy as np

```
arr = np.arange(1,15,3,float)
print(arr)
```

output:

```
[1.4.7.10.13.]
```

Ex:2

```
import numpy as np
    arr = np.arange(10,100,5,int)
print("The array over the given range is ",arr)

over the given range is[10,15,20,25,30,35,40,45,50,55,60,65,70,75,80,90,95]
```

Copy and View

The main difference between a copy and a view of an array is that the copy i s a new array, and the view is just a view of the original array.

Copy:

The copy *owns* the data and any changes made to the copy will not affect ori ginal array, and any changes made to the original array will not affect the c opy

Ex:1

Make a copy,change the orginal array, and display both arrays:

```
import numpy as np
arr=np.array([1,2,3,4,5,6,7,8])
x=arr.copy()
arr[3]=32
print(arr)
print(x)
```

output:

```
[ 1  2  3 32  5  6  7  8]
[1 2 3 4 5 6 7 8]
```

View:

The view *does not own* the data and any changes made to the view will affect the original array, and any changes made to the original array will affect the view.

Ex:1

Make a view, change the original array, and display both arrays:

```
mport numpy as np

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

x = arr.view()

arr[0] = 42


print(arr)

print(x)
```

output:

```
[42 2 3 4 5]

[42 2 3 4 5]
```

Fancy indexing

```
import numpy as np

a = np.arange(1, 10)
print(a)

indices = np.array([2, 3, 4])
print(a[indices])
```

output:

[1,2,3,4,5,6,7,8,9]

[3,4,5]

- Fancy indexing allows you to index an array using another array, a list, or a sequence of integers.

Numeric operations in NumPy are element-wise operations performed on NumPy arrays. These operations include basic arithmetic like addition, subtraction, multiplication, and division, as well as more complex operations like exponentiation, modulus and reciprocal. Here are some examples of these operations:

Addition:
You can add two arrays element-wise using the + operator or the np.add() function.

Ex:1

```
import numpy as np

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

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

result=np.add(a,b)

print(result)

output:

[5 5 5 5 5]
```

Ex:2

```
import numpy as np

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

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

result=(a+b)

print(result)
```

output:

```
[5 5 5 5]
```

Ex:3

Subtraction:

Similarly, subtraction is done using the - operator or np.subtract() function.

```
import numpy as np

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

b = np.array([1,1,1,1])

result = np.subtract(a,b)

print(result)

output:

[0 1 2 3]
```

Ex:4

Multiplication:

For element-wise multiplication, use the * operator or np.multiply().

```
import numpy as np

a = np.array([4,5,6,6])

b = np.array([2,3,4,5])

result=np.multiply(a,b)

print(result)

output:

[ 8 15 24  30]
```

Ex:5

Division:

Element-wise division can be performed with the / operator or np.divide().

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

output:

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

NumPy Broadcasting

in Mathematical operations, we may need to consider the arrays of different shapes. NumPy can perform such operations where the array of different shapes are involved.

see an example:1

Skip Ad

```
array1 = [1, 2, 3]
array2 = [[1], [2], [3]]
```

array1 is a 1-D array and array2 is a 2-D array. Let's perform addition between these two arrays of different shapes.

```
Result = array1+aray2
```

Here, NumPy automatically broadcasts the size of a 1-D array array1 to perform element-wise addition with a 2-D array array2.

```
import numpy as np

# create 1-D array
array1 = np.array([1, 2, 3])

# create 2-D array
array2 = np.array([[1], [2], [3]])

# add arrays of different dimension
# size of array1 expands to match with array2
sum = array1 + array2

print(sum)


output:
[[2 3 4]
 [3 4 5]
 [4 5 6]]
```

Ex:2

```
import numpy as np
arr1=np.array([2,3,4,5])
arr2=np.array([[1],[2],[3],[4],[5]])

sum=arr1+arr2
print(sum)
```

output:

```
[[ 3  4  5  6]
 [ 4  5  6  7]
 [ 5  6  7  8]
 [ 6  7  8  9]
 [ 7  8  9 10]]
```

Broadcasting with Scalars

We can also perform mathematical operations between arrays and scalars (single values). For example,

Ex:1

```
import numpy as np

# 1-D array
array1 = np.array([1, 2, 3])

# scalar
number = 5

# add scalar and 1-D array
sum = array1 + number

print(sum)
```

output:

```
[6 7 8]
```

Ex:2

```
import numpy as np
arr=np.array([4,5,6,7])
#scalar
number=6
sum=arr+number
print(sum)
```

output:

```
[10 11 12 13]
```

Manipulating Arrays in NumPy:

NumPy provides several functions for manipulating arrays, such as adding or r emoving elements, reshaping arrays, and transposing arrays.

Adding Elements to an Array

To add elements to an existing array, you can use the np.append() function. This function takes two arguments, the array and the elements to be added. For example:

Ex:1

```
import numpy as np

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

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

Removing Elements from an Array

To remove elements from an array, you can use the np.delete() function. This function takes two arguments, the array and the indices of the elements to be removed. For example:

```
ex:2
import numpy as np

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

output:
[1 2 3]
```

Reshaping Arrays

To reshape an array, you can use the np.reshape() function. This function takes two arguments, the array and the new shape. For example, to reshape a one-dimensional array into a two-dimensional array:

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

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

Transposing Arrays

To transpose an array, you can use the np.transpose() function. This function takes the array as an argument and returns a new array with the rows and columns interchanged. For example:

Ex:4

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

output:

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

Ex:5

```
import numpy as np
a=np.array([[1,2,4],[3,4,6],[5,6,8]])
b=np.transpose(a)
print(b)
```

output:

```
[[1 3 5]
 [2 4 6]
 [4 6 8]]
```

NumPy Sorting Arrays

Sorting means putting elements in an *ordered sequence.*

Ordered sequence is any sequence that has an order corresponding to elements, like numeric or alphabetical, ascending or descending.

The NumPy ndarray object has a function called sort(), that will sort a specified array.

Ex:1

```
import numpy as np

arr = np.array([2,0,5,1,4])

print(np.sort(arr))
```

output:

```
[0 1 2 4 5]
```

Ex:2

```
import numpy as np

arr = np.array([2,5,8,9,0,1,14,16,11,50,25])

print(np.sort(arr))
```

output:

```
[ 0  1  2  5  8  9 11 14 16 25 50]
```

Sorting a 2-D Array

If you use the sort() method on a 2-D array, both arrays will be sorted:

Ex:3

```
import numpy as np

arr = np.array([[4,2,1,0],[9,10,8,7]])

print(np.sort(arr))
```

output:

```
[[ 0  1  2  4]
 [ 7  8  9 10]]
```

Ex:4

```
import numpy as np

arr = np.array(["cat","fox","bat","ant"])

print(np.sort(arr))
```

output:

```
['ant' 'bat' 'cat' 'fox']
```

NumPy Matrix Operations

Here are some of the basic matrix operations provided by NumPy.

Functions	Descriptions
array()	creates a matrix
dot()	performs matrix multiplication
transpose()	transposes a matrix
linalg.inv()	calculates the inverse of a matrix
linalg.det()	calculates the determinant of a matrix
flatten()	transforms a matrix into 1D array

Create Matrix in NumPy

In NumPy, we use the np.array() function to create a matrix. For example,

Ex:1

```
import numpy as np
```

```
# create a 2x2 matrix
matrix1 = np.array([[1, 3],
                [5, 7]])

print("2x2 Matrix:\n",matrix1)

# create a 3x3  matrix
matrix2 = np.array([[2, 3, 5],
                     [7, 14, 21],
                 [1, 3, 5]])

print("\n3x3 Matrix:\n",matrix2)
```

```
output:
2x2 Matrix:
[[1 3]
 [5 7]]

3x3 Matrix:
 [[ 2  3  5]
 [ 7 14 21]
 [ 1  3  5]]
```

Ex:2

```
import numpy as np
a = np.array([[1,2,3],[4,5,6],[7,8,9]])
print("matrix:\n",a)
```

```
output:
matrix:
 [[1 2 3]
 [4 5 6]
 [7 8 9]]
```

Matrix multiplication

We use the np.dot() function to perform multiplication between two matrices.
import numpy as np

ex:1

```
# create two matrices
matrix1 = np.array([[1, 3],
                         [5, 7]])
```

```
matrix2 = np.array([[2, 6],
                    [4, 8]])

# calculate the dot product of the two matrices
result = np.dot(matrix1, matrix2)

print("matrix1 x matrix2: \n",result)


Output:
matrix1 x matrix2:
[[14 30]
 [38 86]]
```

Ex:2

```
import numpy as np
a=np.array([[1,2],[3,2]])
b=np.array([[4,5],[6,7]])
result=np.dot(a,b)
print("matrix:\n",result)

output:
matrix:
 [[16 19]
 [24 29]]
```

Inverse of a Matrix in

In NumPy, we use the np.linalg.inv() function to calculate the inverse of the given matrix.

However, it is important to note that not all matrices have an inverse. Only square matrices that have a non-zero determinant have an inverse.

Now, let's use np.linalg.inv() to calculate the inverse of a square matrix.

```
import numpy as np

# create a 3x3 square matrix
```

```
matrix1 = np.array([[5, 7, 5],[7, 6, 2],[4, 6, 1]])
# find inverse of matrix1
result = np.linalg.inv(matrix1)
print(result)
```

output:

```
[[-0.08955224  0.34328358 -0.23880597]
 [ 0.01492537 -0.2238806   0.37313433]
 [ 0.26865672 -0.02985075 -0.28358209]]
```

```
import numpy as np
matrix1 = np.array([[1, 3, 5],
                    [7, 9, 2],
                    [4, 6, 8]])
result = np.linalg.inv(matrix1)
print(result)
```

output:

```
[[-1.11111111 -0.11111111  0.72222222]
 [ 0.88888889  0.22222222 -0.61111111]
 [-0.11111111 -0.11111111  0.22222222]]
```

Determinant of a Matrix in NumPy

We can find the determinant of a square matrix using the np.linalg.det() function to calculate the determinant of the given matrix.

Suppose we have a 2x2 matrix A:

a b

c d

So, the determinant of a 2x2 matrix will be:

det(A) = ad - bc

where a, b, c, and d are the elements of the matrix.

Example:

```
import numpy as np
# create a matrix
matrix1 = np.array([[1, 2, 3],
                    [4, 5, 1],
                    [2, 3, 4]])
# find determinant of matrix1
result = np.linalg.det(matrix1)
print(result)
```

output:

```
-5.00
```

Flatten Matrix in NumPy

Flattening a matrix simply means converting a matrix into a 1D array.

To flatten a matrix into a 1-D array we use the array.flatten() function

Ex:

```
import numpy as np
# create a 2x3 matrix
matrix1 = np.array([[1, 2, 3],
                    [4, 5, 7]])
result = matrix1.flatten()
print("Flattened 2x3 matrix:", result)
```

output:

```
Flattened 2x3 matrix: [1 2 3 4 5 7]
```