## Numpy in python:

Python Numpy is a library that handles multidimensional arrays with ease. It has a great collection of functions that makes it easy while working with arrays. Especially with the increase in the usage of Python for data analytic and scientific projects, numpy has become an integral part of Python while working with arrays.

## NumPy – Create 1D Array

One dimensional array contains elements only in one dimension. In other words, the shape of the numpy array should contain only one value in the tuple.

To create a one dimensional array in numpy, you can use either of the **numpy.array()**, **numpy.arange()**, or **numpy.linspace()** functions based on the choice of initialisation.

**Create 1D NumPy Array using array() function**

Numpy array() functions takes a list of elements as argument and returns a one-dimensional array.

In this example, we will import numpy library and use array() function to crate a one dimensional numpy array.

**Python Program**

import numpy as np

#create numpy array

a = np.array([5, 8, 12])

print(a)

**Output**

[ 5, 8, 12]

**Create 1D NumPy Array using arange() function**

NumPy arange() function takes **start**, **end** of a range and the **interval** as arguments and returns a one-dimensional array.

[start, start+interval, start+2\*interval, ... ]

In this example, we will import numpy library and use arange() function to crate a one dimensional numpy array.

**Python Program**

import numpy as np

#create numpy array

a = np.arange(5, 14, 2)

print(a)

**Output**

[ 5, 7, 9, 11, 13]

Array starts with **5** and continues till **14** in the interval of **2**.

**Create 1D NumPy Array using linspace() function**

NumPy linspace() functions takes **start**, **end** and the **number of elements** to be created as arguments and creates a one-dimensional array.

Another very useful method to create NumPy arrays is the linspace method. This method takes three arguments: a start index, end index, and the number of linearly-spaced numbers that you want between the specified range. For instance, if the first index is 1, the last index is 10 and you need 10 equally spaced elements within this range, you can use the linspace method as follows:

lin = np.linspace(1, 10, 10)

The output will return integers from 1 to 10:

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

In this example, we will import numpy library and use linspace() function to crate a one dimensional numpy array.

**Python Program**

import numpy as np

#create numpy array

a = np.linspace(5, 25, 4)

print(a)

**Output**

[ 5. 11.66666667 18.33333333 25. ]

The eye Method:

The eye method can be used to create an identity matrix, which can be very useful to perform a variety of operations in linear algebra. An identity matrix is a matrix with zeros across rows and columns except the diagonal. The diagonal values are all ones. Let's create a 4x4 identity matrix using the eye method:

idn = np.eye(4) print(idn)

array([[1., 0., 0., 0.],

[0., 1., 0., 0.],

[0., 0., 1., 0.],

[0., 0., 0., 1.]])

**NumPy – Create 2D Array**

To create a 2D (2 dimensional) array in Python using NumPy library, we can use any of the following methods.

* **numpy.array()** – Creates array from given values.
* **numpy.zeros()** – Creates array of zeros.
* **numpy.ones()** – Creates array of ones.
* **numpy.empty()** – Creates an empty array.

**Create 2D Array using numpy.array()**

import numpy as np

# create a 2D array with shape (3, 4)

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

print(arr)

**Output**

[[ 1 2 3 4]

[ 5 6 7 8]

[ 9 10 11 12]]

**Create 2D Array using numpy.zeros()**

Pass shape of the required 2D array, as a tuple, as argument to numpy.zeros() function. The function returns a numpy array with specified shape, and all elements in the array initialised to zeros.

import numpy as np

# create a 2D array with shape (3, 4)

shape = (3, 4)

arr = np.zeros(shape)

print(arr)

**Output**

[[0. 0. 0. 0.]

[0. 0. 0. 0.]

[0. 0. 0. 0.]]

**Create 2D Array using numpy.ones()**

Pass shape of the required 2D array, as a tuple, as argument to numpy.ones() function. The function returns a numpy array with specified shape, and all elements in the array initialised to ones.

**Python Program**

import numpy as np

# create a 2D array with shape (3, 4)

shape = (3, 4)

arr = np.ones(shape)

print(arr)

**Output**

[[1. 1. 1. 1.]

[1. 1. 1. 1.]

[1. 1. 1. 1.]]

**Create 2D Array using numpy.empty()**

Pass shape of the required 2D array, as a tuple, as argument to numpy.empty() function. The function returns a numpy array with specified shape.

**Python Program**

import numpy as np

# create a 2D array with shape (3, 4)

shape = (3, 4)

arr = np.empty(shape)

print(arr)

**Output**

[[0. 0. 0. 0.]

[0. 0. 0. 0.]

[0. 0. 0. 0.]]

## NumPy – Create 3D Array

To create a 3D (3 dimensional) array in Python using NumPy library, we can use any of the following methods.

* **numpy.array()** – Creates array from given values.
* **numpy.zeros()** – Creates array of zeros.
* **numpy.ones()** – Creates array of ones.
* **numpy.empty()** – Creates an empty array.

**Create 3D Array using numpy.array()**

Pass a nested list (list of lists of lists) to numpy.array() function.

In the following program, we create a numpy 3D array of shape (2, 3, 4).

**Python Program**

import numpy as np

# create a 3D array with shape (2, 3, 4)

nested\_list = [[[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]],

[[13, 14, 15, 16], [17, 18, 19, 20], [21, 22, 23, 24]]]

arr = np.array(nested\_list)

print(arr)

**Output**

[[[ 1 2 3 4]

[ 5 6 7 8]

[ 9 10 11 12]]

[[13 14 15 16]

[17 18 19 20]

[21 22 23 24]]]

**Create 3D Array using numpy.zeros()**

Pass shape of the required 2D array, as a tuple, as argument to numpy.zeros() function. The function returns a numpy array with specified shape, and all elements in the array initialised to zeros.

**Python Program**

import numpy as np

# create a 3D array with shape (2, 3, 4)

shape = (2, 3, 4)

arr = np.zeros(shape)

print(arr)

**Output**

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

[0. 0. 0. 0.]

[0. 0. 0. 0.]]

[[0. 0. 0. 0.]

[0. 0. 0. 0.]

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

**Create 3D Array using numpy.ones()**

Pass shape of the required 3D array, as a tuple, as argument to numpy.ones() function. The function returns a numpy array with specified shape, and all elements in the array initialised to ones.

**Python Program**

import numpy as np

# create a 3D array with shape (2, 3, 4)

shape = (2, 3, 4)

arr = np.ones(shape)

print(arr)

**Output**

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

[1. 1. 1. 1.]

[1. 1. 1. 1.]]

[[1. 1. 1. 1.]

[1. 1. 1. 1.]

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

**Create 3D Array using numpy.empty()**

Pass shape of the required 3D array, as a tuple, as argument to numpy.empty() function. The function returns a numpy array with specified shape.

**Python Program**

import numpy as np

# create a 3D array with shape (2, 3, 4)

shape = (2, 3, 4)

arr = np.empty(shape)

print(arr)

**Output**

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

[0. 0. 0. 0.]

[0. 0. 0. 0.]]

[[0. 0. 0. 0.]

[0. 0. 0. 0.]

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

**NumPy – Reshape Array:**

To reshape a given array to specific shape using NumPy library, we can use numpy.reshape() function. Pass the given array, and required shape (as tuple) as arguments to the numpy.reshape() function.

Example: Reshape numpy array from (3,4) to (2,6)

In the following program, we reshape a numpy array of shape (3, 4) to (2, 6).

**Python Program**

import numpy as np

# reshape (3, 4) array to (6, 2)

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

shape = (6, 2)

a = np.reshape(arr, shape) print(a)

[[ 1 2] [ 3 4] [ 5 6] [ 7 8] [ 9 10] [11 12]]

Let's create an array of 16 elements using the arange function. Execute the following code:

nums = np.arange(1, 17)

The nums array is a one-dimensional array of 16 elements, ranging from 1 to 16:

array([ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16])

let's convert it into a two-dimensional array of 4 rows and 4 columns:

nums2 = nums.reshape(4, 4)

The array now looks like this:

array([[ 1, 2, 3, 4],

[ 5, 6, 7, 8],

[ 9, 10, 11, 12],

[13, 14, 15, 16]])

NOTE: you cannot reshape an array if the number of elements in the one-dimensional array is not equal to the product of rows and columns of the reshaped array. For instance, if you have 45 elements in a 1-d array, you cannot reshape it into a matrix of 5 row and 10 columns since a 5x10 matrix has 50 elements and the original one only has 45.

**NumPy – Initialize Array with Range of Numbers:**

To initialise an array with a range of numbers in NumPy, you can use numpy.arange() function.

The syntax to call numpy.arange() function is

numpy.arange(start, stop, step)

where

* **start** [mandatory] is the starting of the range.
* **stop** [mandatory] is the ending of the range. stop is not inclusive in the returned array.
* **step** [optional] is the difference between adjacent numbers in the range.

The function returns a numpy array with elements starting from **start**, in steps of step value **step**, until **stop**.

Create 1D numpy array with numbers from 10 to 19

In the following program, we create one dimensional numpy array with a range of numbers from 10 upto 20(20 not included).

**Python Program**

import numpy as np

arr = np.arange(10, 20)

print(arr)

**Output**

[10 11 12 13 14 15 16 17 18 19]

Create numpy array with numbers from 10 to 50 in steps of 5

In the following program, we create one dimensional numpy array with a range of numbers from 10 to 50 in steps of 5.

**Python Program**

import numpy as np

arr = np.arange(10, 50, 5)

print(arr)

**Output**

[10 15 20 25 30 35 40 45]

Create 2D numpy array with a range of numbers

In the following program, we create a two dimensional array with a range of numbers starting from **1**, with an array shape of **(4, 5)**.

Since we need **4\*5=20** elements to create an array of shape **(4, 5)**, and start of the range is **1**, ending of the range must be **(4\*5)+1**.

**Python Program**

import numpy as np

shape = (4, 5)

arr = np.arange(1, (4\*5)+1).reshape(shape)

print(arr)

**Output**

[[ 1 2 3 4 5]

[ 6 7 8 9 10]

[11 12 13 14 15]

[16 17 18 19 20]]

NumPy Array – Iterate over elements

To iterate over elements of a numpy array, you can use **numpy.nditer** iterator object.

**numpy.nditer** provides Python’s standard Iterator interface to visit each of the element in the numpy array. We can use for loop statement to traverse the elements of this iterator object.

Iterate over elements of given 2D numpy array

In the following example, we have a 2D array, and we use **numpy.nditer** to print all the elements of the array.

**Python Program**

import numpy as np

#2D array

a = (np.arange(8)\*2).reshape(2,4)

#print array

print(a)

#iterate over elements of the array

for x in np.nditer(a):

print(x, end=' ')

The nonzero() function is useful to know the positions of elements which are non zero. This function returns an array that contains the indexes of the elements of the array which are not equal to zero.

import numpy

a=array([1,2,0,-1,0,6]) c=numpy.nonzero(a) for i in c: print(i)

(array([0, 1, 3, 5], dtype=int64),)

X = np.array([[1, 0, 0],

[0, 2, 2],

[3, 0, 0]])

print(np.nonzero(X))

(array([0, 1, 1, 2], dtype=int64), array([0, 1, 2, 0], dtype=int64))

The result is a tuple of two NumPy arrays. The first array gives the row indices of non-zero elements. The second array gives the column indices of non-zero elements.

**Get a Row from Numpy Array:**

To get specific row of elements, access the numpy array with all the specific index values for other dimensions andfor the row of elements you would like to get. It is special case of array slicing in Python.

For example, consider that we have a 3D numpy array of shape (m, n, p). And we would like to get the row of elements at ith element along axis=0, and kth element along axis=2. Use the following syntax to get this desired row of elements.

**Access a specific row of elements in given 3D numpy array**

In the following example, we will initialize a 3D array and access a specific row of elements present at index=0 along axis=0, and index=1 along axis=2.

import numpy as np

#initialize an array

arr = np.array([[[11, 11, 9, 9],

[11, 0, 2, 0]],

[[10, 14, 9, 14],

[0, 1, 11, 11]]])

# print shape of array

print('Array Shape: ', arr.shape)

# get the desired row

row = arr[0, :, 1]

print('Desired Row of Elements: ', row)

**Access a specific row or column in 2D Numpy Array**

In the following example, we will initialize a 2D array and access a row and column using array slicing.

**Python Program**

import numpy as np

#initialize an array

arr = np.array([[11, 11, 9, 9],

[11, 0, 2, 0]])

print(arr)

# get index=1 along axis=0 - this means a row in 2D

row = arr[1, :]

print('arr[1, :] : ', row)

# get index=2 along axis=1 - this means a column in 2D

row = arr[:, 2]

print('arr[:, 2] : ', row)

**NumPy – Array Shape or Dimensions:**

To get the shape or dimensions of a NumPy Array, use **ndarray.shape** where ndarray is the name of the numpy array you are interested of. ndarray.shape returns a tuple with dimensions along all the axis of the numpy array.

**Get shape of multi-dimensional NumPy array**

In the following example, we have initialized a multi-dimensional numpy array. Of course, we know the shape of the array by its definition. But, we will use ndarray.shape property to get the shape of the array programmatically.

import numpy as np

#initialize an array

arr = np.array([[[11, 11, 9, 9],

[11, 0, 2, 0]

],

[[10, 14, 9, 14],

[0, 1, 11, 11]]])

# get array shape

shp = arr.shape

print(shp) (2, 2, 4)

**Get shape of 2D NumPy array**

In the following example, we will create a 2D numpy array and find its shape. There are two rows and four columns. So, we should get a tuple of (2, 4). Let us see.

**Python Program**

import numpy as np

#initialize an array

arr = np.array([[11, 11, 9, 9],

[11, 0, 2, 0]])

# get array shape

shape = arr.shape

print(shape) (2, 4)

**Get shape of 1-Dimensional NumPy array**

In the following example, we take a one-dimensional numpy array, and find its shape.

**Python Program**

import numpy as np

#initialize an array

arr = np.array([11, 11, 9, 9])

# get array shape

shape = arr.shape

print(shape) # There are four elements, and shape will be a tuple with 4

**Get Array Size – NumPy:**

In NumPy, array size is defined as the total number of elements in the array. To get the size of a numpy array, we can read the **size** property of the array object.

For example, if **arr** is the numpy array, then **arr.size** returns the size of the array.

**Get size of an array with shape (3, 5)**

In the following program, we take a numpy array of shape (3, 5), and get its size programmatically using **size** property.

**Python Program**

import numpy as np

arr = np.ones((3, 5))

arr

size = arr.size

print(size)

**Get size of an array with shape (3, 4, 3):**

In the following program, we take a numpy array of shape (3, 4, 3), and get its size programmatically using **size** property.

arr = np.ones((3, 4, 3))

arr size=arr.size print(size)

**NumPy Array – Iterate over elements:**

To iterate over elements of a numpy array, you can use **numpy.nditer** iterator object.

**numpy.nditer** provides Python’s standard Iterator interface to visit each of the element in the numpy array. We can use For loop statement to traverse the elements of this iterator object.

Iterate over elements of given 2D numpy array

In the following example, we have a 2D array, and we use **numpy.nditer** to print all the elements of the array.

**Python Program**

import numpy as np

#2D array

a = (np.arange(8)\*2).reshape(2,4)

#print array

print("The array\n",a)

print("\nIterating over all the elemnets of array")

#iterate over elements of the array

for x in np.nditer(a):

print(x, end=' ')

import numpy as np

# Create a NumPy array

arr = np.array([20, 35, 40, 25, 50])

# Iterate over an array using for loop

for x in arr:

print(x)

Iterate Index & Value of Array Using for Loop

We can also iterate both indexes and values of a given array using for loop and np.ndenumerate() function. For example,

# Iterate by getting index and value

for index, value in np.ndenumerate(arr):

print(index, value)

Iterate 2-Dimensional Array using For Loop:

Let’s create a 2-Dimensional NumPy array using np. arange() and np.reshape() functions and then, iterate it using for loop with nditer() function. The below syntax will iterate both indexes and values of the given array and return them sequentially.

arr = np.arange(12)

arr1 = arr.reshape(3,4)

print(arr1)

# Iterate two dimensional array

print("Output:")

for x in np.nditer(arr1):

print(x)

**Iterate Indexes & Values of 2- Dimensional Array using For Loop:**

Let’s also see how to iterate over both indexes and values of a given 2-D array using Python for loop along with the np.ndenumerate() function. For example,

# Iterate 2-D array and get indexes & values

for index, value in np.ndenumerate(arr):

print(index, value)

**NumPy – Concatenate Arrays:**

To concatenate arrays in NumPy, call numpy.concatenate() function, pass tuple of arrays to be concatenated, and the axis along which concatenation must happen, as arguments.

The syntax to concatenate **array1** and **array2** numpy arrays is

numpy.concatenate((array1, array2, ...), axis=0)

By default, the arrays are concatenated along axis=0. We can override this behaviour by passing a specific value for axis parameter.

**Note:** The arrays to be concatenated must have the same shape along all but the specified axis.

Concatenate arrays along default axis (=0)

In the following program, we take two numpy arrays. of shape (3, 2) and (4, 2). We concatenate these two array along default axis.

**Python Program**

import numpy as np

array1 = np.array([[1, 2], [3, 4], [5, 6]])

array2 = np.array([[7, 8], [9, 10], [11, 12]])

output = np.concatenate((array1, array2))

print(output)

**Output**

[[ 1 2]

[ 3 4]

[ 5 6]

[ 7 8]

[ 9 10]

[11 12]]

For the two arrays, the length of the arrays along axis=0 is different, but the length of the arrays along axis=1 is same which satisfies the required condition for concatenation.

Concatenate arrays along axis=1

In the following program, we take a numpy array of shape (3, 4), and (3, 2), and concatenate them along axis=1.

**Python Program**

import numpy as np

array1 = np.array([[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]])

array2 = np.array([[13, 14], [15, 16], [17, 18]])

output = np.concatenate((array1, array2), axis=1)

print(output)

**Copy Array using numpy.copy():**

In the following example, we will copy the elements of an array **a** to another array **b**.

**Python Program**

import numpy as np

# create a numpy array

a = np.array([[8, 2, 3],

[4, 7, 6]])

# copy contents of a to b

b = a.copy()

# modify a

a[1, 2] = 13

# check if b has remained the same

print(a)

print(b)

**Output:**

a

[[ 8 2 3]

[ 4 7 13]]

b

[[8 2 3]

[4 7 6]]

Even if we have changed the contents of **a**, the contents of **b** are unaffected.

**Convert Numpy Array to List:**

**NumPy – Convert Array to a Python List**

To convert a Numpy Array into List in Python, call **numpy.tolist()** function and pass the numpy array as argument to this function.

The syntax to call **numpy.tolist()** to convert a numpy array into list is

numpy.tolist(arr)

where **arr** is a numpy array.

**Return Value: numpy.tolist()** returns an object of type **list**.

**Convert 1D numpy array to a list**

In this example, we take a numpy array in **arr**, and convert this numpy array into list using numpy.tolist().

**Python Program**

import numpy

arr = numpy.array([1, 2, 3, 4, 5, 10, 15])

x = arr.tolist()

print(arr)

print(x)

**Convert 2D numpy array to a list**

If the numpy array has more than one-dimension, then the returned list is a nested list.

In the following program, we take a 2D numpy array in **arr**, and convert this numpy array into a list using numpy.tolist().

**Python Program**

import numpy

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

x = arr.tolist()

print(arr)

print(x)

**Output**

Numpy Array:

[[1 2]

[3 4]]

List:

[[1, 2], [3, 4]]

**Convert List to Numpy Array**

To convert a List into Numpy Array in Python, call **numpy.array()** function and pass the list as argument to this function.

The syntax to call numpy.array() to convert a list into numpy array is

numpy.array(x)

where **x** is a list.

**Return Value: numpy.array()** returns a numpy array of type **numpy.ndarray**.

**Convert a list of numbers to a numpy array:**

In this example, we take a list in **x**, and convert this list into numpy array using numpy.array().

**Python Program**

import numpy

x = list([1, 2, 4, 8, 16, 32])

arr = numpy.array(x)

print("List :", x)

print("Numpy Array :", arr)

**Output:**

List: [1, 2, 4, 8, 16, 32]

Numpy Array: [ 1 2 4 8 16 32]

**Convert a list of lists to a numpy array**

In this example, we take a list of lists in **x**, and convert this list into numpy array using numpy.array().

**Python Program**

import numpy

x = list([[1, 2, 4], [8, 16, 32]])

arr = numpy.array(x)

print("List :", x)

print("Numpy Array :", arr)

**Output**

List : [[1, 2, 4], [8, 16, 32]]

Numpy Array : [[ 1 2 4]

[ 8 16 32]]

Since, there are two levels of lists, we get a two-dimensional numpy array.

**NumPy – Select Rows / Columns by Index:**

import numpy as np

# Create a 2D Numpy arr with 3 rows & 3 columns | Matrix

arr = np.array(([21, 22, 23], [11, 22, 33], [43, 77, 89]))

Output:

[[21 22 23]

[11 22 33]

[43 77 89]]

Select a single element from 2D Numpy Array by index

We can use [][] operator to select an element from Numpy Array i.e.

arr[row\_index][column\_index]

**Example 1:**

Select the element at row index 1 and column index 2.

# Select element at row index 1 & column index 2

num = arr[1][2]

Output:

33 #element at row index 1 & column index 2 is

**Example 2:**

Or we can pass the comma separated list of indices representing row index & column index too i.e.

# Another way to select element at row index 1 & column index 2

num = arr[1, 2]

Output: 33 # element at row index 1 & column index 2

**Select Rows by Index from a 2D Numpy Array:**

We can call [] operator to select a single or multiple row. To select a single row use, arr[row\_index]

It will return a complete row at given index.

**To select multiple rows use,**

arr[start\_index: end\_index, :]

It will return rows from start\_index to end\_index – 1 and will include all columns.

arr = np.array(([21, 22, 23], [11, 22, 33], [43, 77, 89]))

Output:

[[21 22 23]

[11 22 33]

[43 77 89]]

Let’s **select a row** at index 1 i.e.

# Select a Row at index 1

row = arr[1]

print('Contents of Row at Index 1: ' , row)

Output: Contents of Row at Index 1: [11 22 33]

**Select multiple rows** from index 1 to 2 i.e.

# Select multiple rows from index 1 to 2

rows = nArr2D[1:3, :]

print('Rows from Index 1 to 2 :')

print(rows)

Output: Rows from Index 1 to 2:

[[11 22 33]

[43 77 89]]

**Select multiple rows** from index 1 to last index

# Select multiple rows from index 1 to last index

rows = arr[1: , :]

print('Rows from Index 1 to last row :')

print(rows)

Output:

[[11 22 33]

[43 77 89]]

**Select Columns by Index from a 2D Numpy Array**

To **select a single column** use,

arr[ : , column\_index]

It will return a complete column at given index.

To **select multiple columns** use,

arr[ : , start\_index: end\_index]

It will return columns from start\_index to end\_index – 1.

**Select a column** at index 1

# Select a column at index 1

column = nArr2D[:, 1]

print('Contents of Column at Index 1: ', column)

Output: Contents of Column at Index 1: [22 22 77]

**Select multiple columns** from index 1 to 2

# Select multiple columns from index 1 to 2

columns = nArr2D[: , 1:3]

print('Column from Index 1 to 2 :')

print(columns)

Output:

Column from Index 1 to 2 :

[[22 23]

[22 33]

[77 89]]

**Select multiple columns** from index 1 to last index

# Select multiple columns from index 1 to last index

columns = arr[:, 1:]

Output is same as above because there are only 3 columns 0,1,2. So 1 to last columns means columns at index 1 & 2.

**Select a Sub Matrix or 2d Numpy Array from another 2D Numpy Array**

To select sub 2d Numpy Array we can pass the row & column index range in [] operator i.e.

arr[start\_row\_index : end\_row\_index , start\_column\_index : end\_column\_index]

It will return a sub 2D Numpy Array for given row and column range.

Select a sub 2D Numpy Array from row indices 1 to 2 & column indices 1 to 2

# Select a sub 2D array from row indices 1 to 2 & column indices 1 to 2

sub2DArr = arr[1:3, 1:3]

print(sub2DArr)

Output: Sub 2d Array :

[[22 33]

[77 89]]

Now modify the contents of row i.e.

# Change all the elements in selected sub array to 100

row[:] = 100

New contents of the row will be

[100 100 100]

print(arr)

Output:

array([[ 21, 22, 23],

[100, 100, 100],

[ 43, 77, 89]])

arr[0,:1]=50 # To modify/change only one element.

print(arr)

array([[ 50, 22, 23], [100, 100, 100], [ 43, 77, 89]])

Write a NumPy program to compute the inverse of a given matrix.

import numpy as np

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

print("Original matrix:")

print(m)

result = np.linalg.inv(m)

print("Inverse of the said matrix:")

print(result)

Original matrix:

[[1 2]

[3 4]]

Inverse of the said matrix:

[[-2. 1. ]

[ 1.5 -0.5]]

**Explanation:**

m = np.array([[1,2],[3,4]]): This statement creates a 2x2 NumPy array m with the specified elements.

result = np.linalg.inv(m): This line computes the inverse of the matrix m. The inverse of a square matrix A (if it exists) is another matrix, denoted as A^(-1), such that their product results in the identity matrix (A \* A^(-1) = I). In this case, the inverse of the 2x2 matrix m is calculated as follows:

1/(ad-bc) \* [[d, -b], [-c, a]]

where a, b, c, and d are the elements of the matrix, and ad-bc is the determinant.

For the given matrix m, the inverse is calculated as:

1/((1\*4)-(2\*3)) \* [[4, -2], [-3, 1]] 1/(-2) \* [[4, -2], [-3, 1]] -0.5 \* [[4, -2], [-3, 1]]

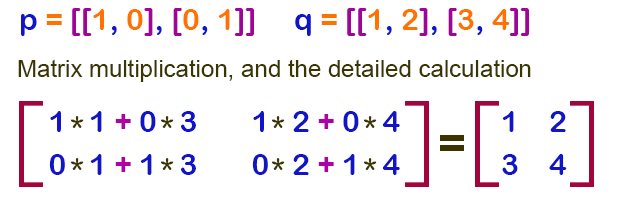
resulting in the inverse matrix [[-2., 1.], [1.5, -0.5]].

**Compute the multiplication of two given matrixes:**

P = [[1, 0], [0, 1]]

Q = [[1, 2], [3, 4]]

**Matrix multiplication, and the detailed calculation:**

****

import numpy as np

p = [[1, 0], [0, 1]]

q = [[1, 2], [3, 4]]

print("original matrix:")

print(p)

print(q)

result1 = np.dot(p, q)

print("Result of the said matrix multiplication:")

print(result1)

Output:

original matrix:

[[1, 0], [0, 1]]

[[1, 2], [3, 4]]

Result of the said matrix multiplication:

[[1 2]

[3 4]]

**Explanation:**

p = [[1, 0], [0, 1]]

q = [[1, 2], [3, 4]]

is calculated as follows:

result1[0,0] = (p[0,0] \* q[0,0]) + (p[0,1] \* q[1,0]) = (1 \* 1) + (0 \* 3) = 1

result1[0,1] = (p[0,0] \* q[0,1]) + (p[0,1] \* q[1,1]) = (1 \* 2) + (0 \* 4) = 2

result1[1,0] = (p[1,0] \* q[0,0]) + (p[[[1, 2], [3, 4]]

At first two 2x2 matrixes p and q have been declared.

result1 = np.dot(p, q) This code calculates the matrix multiplication (dot product) of p and q. The dot product 1,1] \* q[1,0]) = (0 \* 1) + (1 \* 3) = 3

result1[1,1] = (p[1,0] \* q[0,1]) + (p[1,1] \* q[1,1]) = (0 \* 2) + (1 \* 4) = 4

Finally print() function prints the resulting 2x2 matrix.

**NumPy Dot Product – numpy.dot()**

**NumPy Dot Product**

To compute dot product of numpy nd arrays, you can use **numpy.dot()** function. **numpy.dot()** function accepts two numpy arrays as arguments, computes their dot product, and returns the result.

**Syntax:**

The syntax of numpy.dot() function is

numpy.dot(a, b, out=None)

| **Parameter** | **Description** |
| --- | --- |
| a | [mandatory] First argument for dot product operation. |
| b | [mandatory] Second argument for dot product operation. |
| out | [optional] This argument is used for performance. This has to be a C-contiguous array, and the dtype must be the dtype that would be returned for dot(a,b). |

## Behavior of numpy.dot() based on Input Array Dimensions

The following table specifies the type of operation done based on the dimensions of input arrays: a and b.

| **Dimension of ‘a’ and ‘b’** | **Output** |
| --- | --- |
| Zero-Dimension (Scalar) | Multiplication of two scalars, a and b. |
| One-Dimensional Arrays (Vector) | Inner product of vectors. |
| Two-Dimensional Arrays (Matrix) | Matrix Multiplication. |
| a: N-Dimensional Array b: 1-D Array | Sum product over the last axis of a and b. |
| a: N-Dimensional Array b: M-Dimensional Array (M>=2) | Sum product over the last axis of a and second-to-last axis of b. |

**Examples**

**Dot Product of Scalars**

In this example, we take two scalars and calculate their dot product using numpy.dot() function. Dot product using numpy.dot() with two scalars as arguments return multiplication of the two scalars.

# **Python Program**

import numpy as np

a = 3

b = 4

output = np.dot(a, b)

print(output)

**Output**

12

**Explanation**

output = a \* b

= 3 \* 4

= 12

**Dot Product of 1D Arrays (Vectors)**

In this example, we take two numpy one-dimensional arrays and calculate their dot product using numpy.dot() function. We already know that, if input arguments to dot() method are one-dimensional, then the output would be inner product of these two vectors (since these are 1D arrays).

# **Python Program**

import numpy as np

#initialize arrays

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

B = np.array([3, 4, 7, 8])

#dot product

output = np.dot(A, B)

print(output)

**Output**

77

**Dot Product**

output = [2, 1, 5, 4].[3, 4, 7, 8]

= 2\*3 + 1\*4 + 5\*7 + 4\*8

= 77

### Dot Product of 2-D Arrays (Matrix)

In this example, we take two two-dimensional numpy arrays and calculate their dot product. Dot product of two 2-D arrays returns matrix multiplication of the two input arrays.

# **Python Program**

import numpy as np

#initialize arrays

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

B = np.array([[3, 4], [7, 8]])

#dot product

output = np.dot(A, B)

print(output)

**Output**

[[13 16]

[43 52]]

**Dot Product**

output = [[2, 1], [5, 4]].[[3, 4], [7, 8]]

= [[2\*3+1\*7, 2\*4+1\*8], [5\*3+4\*7, 5\*4+4\*8]]

= [[13, 16], [43, 52]]

**Cross Product – NumPy:**

NumPy – Cross Product of Arrays

Cross product of two vectors yield a vector that is perpendicular to the plane formed by the input vectors and its magnitude is proportional to the area spanned by the parallelogram formed by these input vectors.

In this tutorial, we shall learn how to compute cross product using numpy.cross() function.

Examples

Cross product of vectors in 2D plane

In this example, we shall take two points in XY plane as numpy arrays and find their cross product.

# **Python Program**

import numpy as np

#initialize arrays

A = np.array([2, 3])

B = np.array([1, 7])

#compute cross product

output = np.cross(A, B)

print(output)

11

**Mathematical Proof**

cross(A,B) = 2\*7 - 3\*1

= 11

Consider that vectors [2,3] and [1,7] are in [X,Y] plane. Then the cross product [11] is in the axis perpendicular to [X, Y], say Z with magnitude 11.

**Cross product of vectors in 3D plane**

In this example, we shall take two NumPy Arrays, each of length 3 (representing a point in 3D space), and find their cross product.

# **Python Program**

import numpy as np

#initialize arrays

A = np.array([2, 7, 4])

B = np.array([3, 9, 8])

#compute cross product

output = np.cross(A, B)

print(output)

[20 -4 -3]

**Mathematical Proof**

cross(A, B) = [(7\*8-9\*4), -(2\*8-4\*3), (2\*9-7\*3)]

= [20, -4, -3]

Output vector [20, -4, -3] is perpendicular to the plane formed by the input vectors [2, 7, 4], [3, 9, 8].

**numpy.where():**

The where() function returns the indices of elements in an input array where the given condition is satisfied.

import numpy as np

x = np.arange(9.).reshape(3, 3)

print(x)

print 'Indices of elements > 3'

y = np.where(x > 3)

print(y)

print 'Use these indices to get elements satisfying the condition'

print(x[y])

[[ 0. 1. 2.]

[ 3. 4. 5.]

[ 6. 7. 8.]]

Indices of elements > 3

(array([1, 1, 2, 2, 2]), array([1, 2, 0, 1, 2]))

Use these indices to get elements satisfying the condition

[ 4. 5. 6. 7. 8.]

## numpy.extract()

The **extract()** function returns the elements satisfying any condition.

import numpy as np

x = np.arange(9.).reshape(3, 3)

print(x)

# define a condition

condition = np.mod(x,2) == 0

print 'Element-wise value of condition'

print(condition)

print 'Extract elements using condition'

print (np.extract(condition, x))

It will produce the following output −

[[ 0. 1. 2.]

[ 3. 4. 5.]

[ 6. 7. 8.]]

Element-wise value of condition

[[ True False True]

[False True False]

[ True False True]]

Extract elements using condition

[ 0. 2. 4. 6. 8.]