numpy module

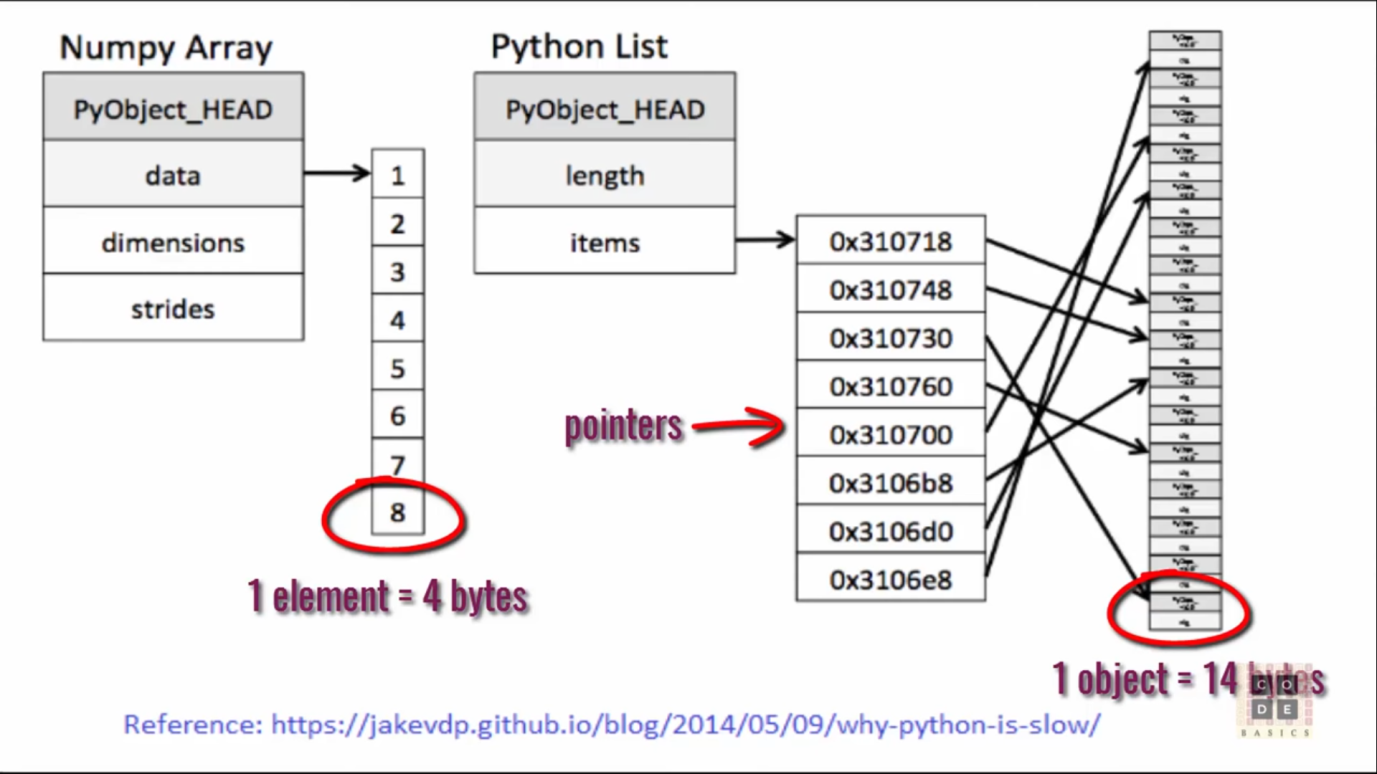
* numpy is used to work with arrays.
* An array is very much similar to a list. We can access array elements with an index.
* The advantages of an array over a list is:

-> needs less memory

-> convenient

-> fast

To compare the memory occupied by list and array:



Array stores elements directly, where as in lists , every element is an object. It stores the address of those objects which inturn point to the data.

* Note:

To get the memory occupied by a list in bytes:

import sys

list1=range(1000)

print sys.getsizeof(list1)

To get memory occupied by an array:

import numpy

arr=numpy.arange(1000) #arange() is just like range()

print arr.size \* arr.itemsize

--> arr.size gives the length of the array and arr.itemsize gives the size of every element of the array.

Numpy elements are convenient to use compared to lists

Note:

zip(list1,list2): produces a tuple , where ith element from list1 and ith element from list2 become the ith element in the result.

Ex:

import sys

import time

list1=range(10)

list2=range(100,111)

print zip(list1,list2)

ouput:

[(0, 100), (1, 101), (2, 102), (3, 103), (4, 104), (5, 105), (6, 106), (7, 107), (8, 108), (9, 109)]

Ex:

import sys

import time

list1=range(10)

list2=range(100,111)

start=time.time()

for (x,y) in zip(list1,list2):

print x+y

end=time.time()

print "time took=",(end-start)\*1000

ouput: 100, 102, 104, 106

108

110

112

114

116

118

Ex: Same program with numpy array

import time

import numpy

start=time.time()

a1=numpy.arange(1000)

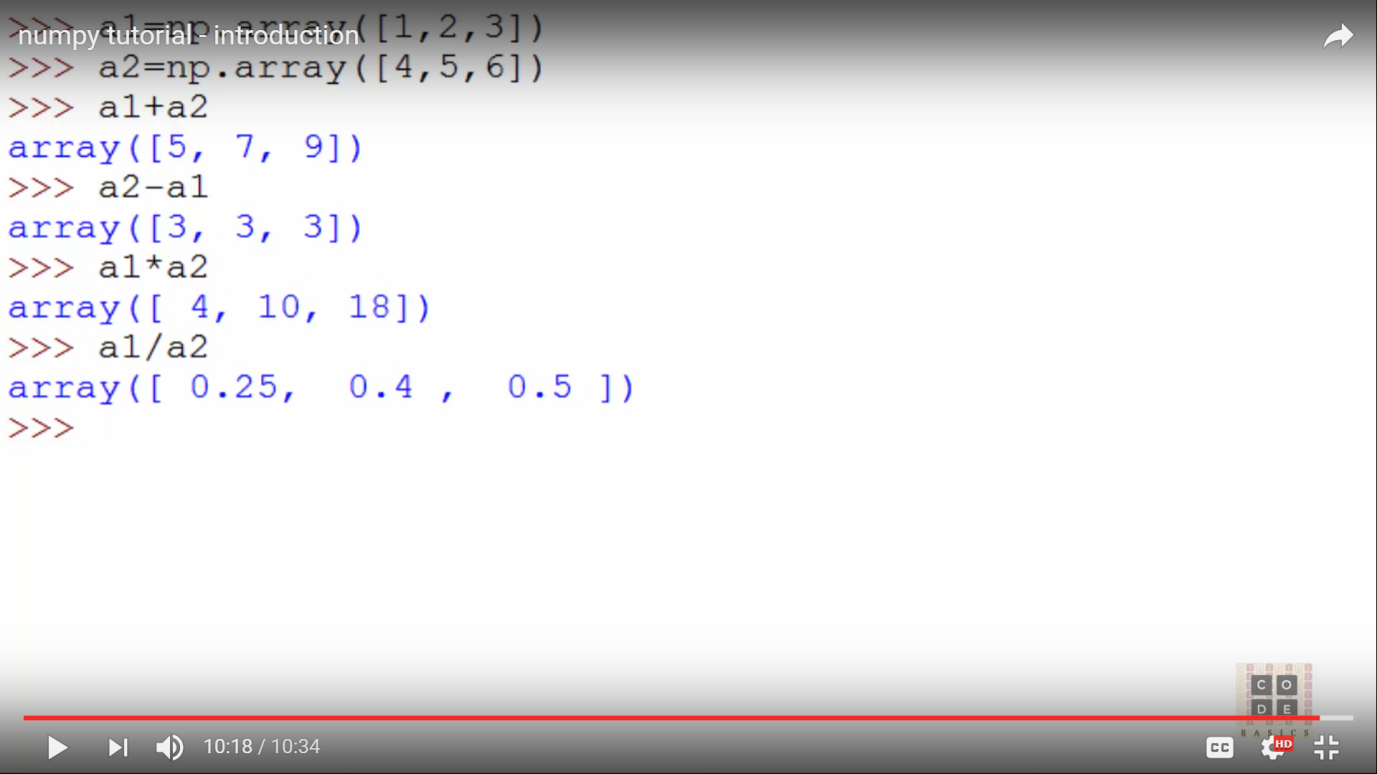
a2=numpy.arange(1000)

result=a1+a2

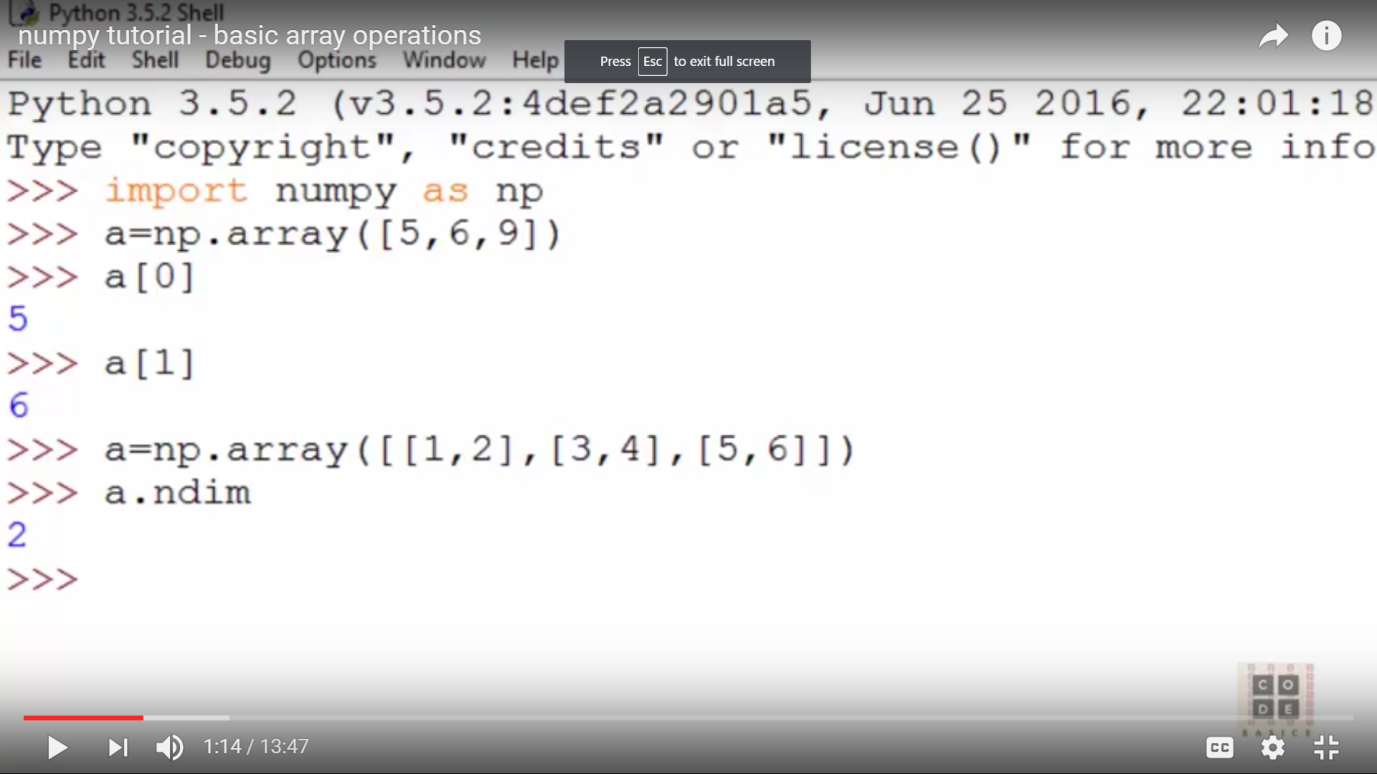
end=time.time()

print "time took=",(end-start)\*1000

Array operations:



Creating multi dimenstional array



-> ndim gives the dimensions of the array.

--> a1.itemsize: size occupied by every element in bytes. (int:4 bytes, float: 8 bytes for 64-bit)

--> a1.dtype: current data type of elements stored in array.

--> a1.size: no.of elements.

--> a1.shape: gives (no. of rows, no. of cols). above example, output will be (3,2)

--> To specify data type for the array,

a1=numpy.array([[1,2],[3,4],[5,6]],dtype=float64/int64/complex)

--> To initialize array with zeroes,

numpy.zeros([no.of rows,no.of cols]):

Ex:numpy.zeros([3,4])

[[0,0,0,0],

[0,0,0,0],

[0,0,0,0],

[0,0,0,0]]

--> To initalize array with ones,

numpy.ones(rows,cols)

--> arange(min,max,step): similar to range()

numpy.arange(5,10,2)

--> numpy.linspace(min,max,howmany nums)

Ex: numpy.linspace(1,5,10): generates 10 numbers between 1 and 5 (float nums)

--> To change existing array dimensions,

a1.reshape(2,3): changes existing shape for 2 rows and 3 cols

--> a1.ravel(): converts array into one dimension. won't change original array

--> a1.min(): prints min element in the array

--> a1.max(): prints max array element

--> a1.sum(): sums all the numbers of array.

--> a1.sum(axis=0): sums all elements column wise(sum of elements in col 0, col1 ...)

--> a1.sum(axis=1): sum of elements row wise

--> numpy.sqrt(a1): finds square root of every element

--> we can add elements of two arrays :

Ex: a=numpy.array([[1,2],[3,4]])

b=numpy.array([[5,6],[7,8])

a+b

[[6,8],[10,12]]

--> a\*b

--> a/b

--> a.dot(b): matrix product

Indexing and Slicing

a=numpy.array([6,7,8])

-->print a[0:2] --> prints 0,1 elements

-->a[-1]: first element from the end

Ex:

a=numpy.array([[6,7,8],

[1,2,3],

[9,3,2]])

--> print a[1,2]: 1st row, 2nd col

3

--> a[0:2,2]

goes through 0th row and first col and prints 2nd element there

o/p: [8,3]

--> print a2[0:2,0:2]: goes through oth row and 1 st row and print 1st and 2nd elements in each row.

--> a[-1]: last element

o/0: [9,3,2]

--> a[-1,0:2]: in the last row, print 0,1 elements

o/p: [9,3]

--> a[:,1:3] : goes through all the rows and print 1st and 2nd elements

[[7,8],[2,3],[3,2])

Iteration through an array:

import numpy as np

a=np.array([[6,7,8],[1,2,3],[9,3,2]])

for row in a:

print row

o/p: [6 7 8]

[1 2 3 ]

[9 3 2]

Ex:

for cell in a.flat:

print cell

--> flattens the array

o/p: 6

7

8

1

2

3

9

3

2

--> Ex:

a=[[0,1],[2,3],[4,5]]

b=[[6,7],[8,9],[10,11]]

np.vstack((a,b))

merges one array and other vertically

o/p: [[0,1]

[2,3]

[4,5]

[6,7]

[8,9]

[10,11])

--> np.hstack((a,b)): horizontal stacking

0/p:[[0,1,6,7],

[2,3,8,9],

[4,5,10,11]]

--> splitting an array

a=np.range(30).reshape(2,15)

o/p: [[0,1,2,3,4,5,6,7,8,9,10,11,12,13,14],

[15,16,17,18,19,20,21,22,23,24,25,26,27,28,29]]

np.hsplit(a,3):

splits into 3 parts horizontally

o/p: array([[0,1,2,3,4],

[15,16,17,18,19]]) -->part1

array([[5,6,7,8,9],

[20,21,22,23,24]]) --> part2

array([[10,11,12,13,14],[25,26,27,28,29]))] --> part3

np.vsplit(a,2): splits vertically into two parts

o/p:

array([[0,1,2,3,4,5,6,7,8,9,10,11,12,13,14]])

array([[15,16,17,18,19,20,21,22,23,24,25,26,27,28,29]])

Boolean indexing

a=np.arange(12).reshape(3,4)

o/p:

array([[0,1,2,3],

[4,5,6,7],

[8,9,10,11]])

b=a>4

b will be :

array([[False,False,False,False],

[F,T,T,T],

[T,T,T,T]], dtype=bool)

---->

a[b]: in array 'b', where ever there is True, it returns the corresponding elements from the array 'a'

o/p: [5,6,7,8,10,11]

indirectly, we are extracting all elements which are greater than 4 from the original array.

--> a[b]=-1

Replaces every element greater than 4 with -1

Now a is:

[[0,1,3,4],

[4,-1,-1,-1],

[-1.-1,-1,-1]]

--> Inserting new elements into an array:

import numpy

a=[[0,1],[2,3],[4,5]]

a.insert(0,[100,200])

print a

--->

a.append([100,200])

--->

a.remove([0,1])

-->

a.count([0,1])

---> extend()

import numpy

a=[[0,1],[2,3],[4,5]]

b=[1,2,3]

a.extend(b)

print a

o/p:

[[0, 1], [2, 3], [4, 5], 1, 2, 3]

--> other operations

import numpy

a=[[0,1],[2,3],[4,5]]

print a.index([0,1])

a.pop()

print a

a.reverse()

print a

o/p: 0

[[0, 1], [2, 3]]

[[2, 3], [0, 1]]

--->sort():

import numpy

a=[[0,100],[20,3],[4,5]]

a.sort()

print a

o/p: [[0, 100], [4, 5], [20, 3]]

-->

Numpy Vs lists

* Lists are slow to process compared to numpy arrays.
* Numpy stands for “numerical python”.
* They are 50 times faster than lists.

Why numpy is faster than lists?

* Numpy arrays are stored in continuous place in memory, where as lists store at different locations.

Ex: Creating numpy arrays using a list

import numpy

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

print(arr)

o/p: [ 1 2 3 4 5]

We can pass a list, tuple or an array object to “array()” method and it will be converted to an numpy array.

Ex: Creating array using a tuple

import numpy

arr=numpy.array((1,2,3,4,5))

print(type(arr))

o/p:

[1 2 3 4 5]

<class numpy.ndarray>

Nested arrays

Ex: 0-D arrays

import numpy

arr=numpy.array(42)

print(arr)

o/p: 42

Ex: 1-D array

An array containing many 0-D elements is an 1-D array

import numpy

arr=numpy.array([10,20,30])

print(arr)

o/p: [10,20,30]

Ex: 2-D array (or) matrix

An array containing 1-D elements is called 2-D array

import numpy

arr=numpy.array([[10,20,30],[100,200,300]])

print(arr)

o/p:

[[ 10 20 30]

[100 200 300]]

Ex: 3-D arrays

An array containing 2-D elements is called a 3-D array

import numpy

arr=numpy.array([[[10,20,30],[100,200,300]],

[[1,2,3],[1000,2000,3000]]])

print(arr)

o/p:

[[[ 10 20 30]

[ 100 200 300]]

[[ 1 2 3]

[1000 2000 3000]]]

Ex: To check the no. Of dimensions,

print(arr.ndim)

Numpy array indexing

Index starts from 0.

Ex: print(arr[0])

Print(arr[0]+arr[2]) 🡪 adds two elements

Access 2-D arrays:

import numpy

arr=numpy.array([[1,2,3,4,5],[90,89,56,34,45]])

print(arr[0])

print(arr[0,2]) #3rd element of the 1st array

print(arr[1,2]) #3rd element of the second array

o/p:

[1 2 3 4 5]

3

56

Accessing 3-D arrays:

import numpy as np

arr=np.array([[[1,2,3],[10,20,30],[100,200,300]]])

print(arr[0,1,2])

o/p: 30

Negative indexing:

import numpy as np

arr=np.array([[[1,2,3],[10,20,30],[100,200,300]]])

print(arr[0,1,-2])

o/p:

20

(i.e. 0th index, 1 st element (10,20,30), 2nd element from the end(i.e.20)

Numpy Array Slicing

[start:end]

[start:end:step]

Ex:

import numpy as np

arr=np.array([10,20,30,40,50])

print(arr[0:4])

print(arr[0:4:2]) #indices 0 to 3, every 2nd element

print(arr[:4]) #indices 0 to 3

print(arr[2:]) #indices 2 to end

o/p:

[10 20 30 40]

[10 30]

[10 20 30 40]

[30 40 50]

Negative Slicing

import numpy as np

arr=np.array([10,20,30,40,50])

print(arr[-3:-1]) #start at -3, stop at 1st element from the last

o/p:

[30 40]

Ex:

import numpy as np

arr=np.array([10,20,30,40,50])

print(arr[::2]) #start at 0, till the end, increment is 2i.e 0,2,4

o/p: [10 30 50]

Slicing 2-D arrays

Ex:

import numpy as np

arr=np.array([[10,20,30,40,50],[100,200,300,400,500]])

print(arr.ndim)

print(arr[::2]) # 0 to end, but every 2nd element. So 2nd element [100,200,300,400,500] is skipped.

o/p:

2

[[10 20 30 40 50]]

Ex:

import numpy as np

arr=np.array([[10,20,30,40,50],[100,200,300,400,500]])

print(arr[1,1:4]) #1st element🡪[100,200,300,400,500]. Out of it, eles from index 1,2,3 are accessed

o/p:

[200 300 400]

Ex:

import numpy as np

arr=np.array([[10,20,30,40,50],[100,200,300,400,500]])

print(arr[0:2,2]) #elements at index 2 from both the arrays

o/p:

[30 300]

Ex:

import numpy as np

arr=np.array([[10,20,30,40,50],[100,200,300,400,500]])

print(arr[0:2,1:4]) # arrays 0,1..out of them eles at the index 1,2,3

o/p:

[[20 30 40

[200 300 400]]

Numpy data types

By default Python have these data types:

strings - used to represent text data, the text is given under quote marks. eg. "ABCD"

integer - used to represent integer numbers. eg. -1, -2, -3

float - used to represent real numbers. eg. 1.2, 42.42

boolean - used to represent True or False.

complex - used to represent a number in complex plain. eg. 1.0 + 2.0j, 1.5 + 2.5j

NumPy has some extra data types, and refer to data types with one character, like i for integers, u for unsigned integers etc.

Below is a list of all data types in NumPy and the characters used to represent them.

i - integer

b - boolean

u - unsigned integer

f - float

c - complex float

m - timedelta

M - datetime

O - object

S - string

U - unicode string

V - fixed chunk of memory for other type ( void )

To check the data type of an array, “dtype” can be used.

Ex:

import numpy as np

arr=np.array([[10,20,30,40,50],[100,200,300,400,500]])

print(arr.dtype)

o/p:

int32

Ex:

import numpy as np

arr=np.array(["apples","boy","cat"])

print(arr.dtype)

o/p:

<U6

Creating arrays with a selected data type

import numpy as np

arr=np.array([1000,2000,3000],dtype="i")

print(arr.dtype)

o/p:int32

Ex: creating an array with 4byte integer data type.

import numpy as np

arr=np.array([1000,2000,3000],dtype="i")

print(arr.dtype)

o/p: int32

Converting data types of existing arrays

import numpy as np

arr=np.array([1.2,2.3,3.4])

print(arr.dtype)

arr2=arr.astype('i') #copy the old array into new array, also change its type

print("new array's type is",arr2.dtype)

print(arr2)

o/p:

float64

new array's type is int32

[1 2 3]

Ex:

import numpy as np

arr=np.array([1.2,2.3,3.4])

print(arr.dtype)

arr2=arr.astype('bool') #copy the old array into new array, also change its type

print("new array's type is",arr2.dtype)

print(arr2)

o/p:

float64

new array's type is bool

[ True True True]

Copy Vs View on numpy arrays

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

The copy owns the data and any changes made to the copy will not affect original array, and any changes made to the original array will not affect the copy.

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:

import numpy as np

arr=np.array([10,20,30,40,50])

arr2=arr.copy()

arr2[2]=1000 #will not effect the original array

print(arr)

print(arr2)

o/p:

[10 20 30 40 50]

[ 10 20 1000 40 50]

Ex:

import numpy as np

arr=np.array([10,20,30,40,50])

arr2=arr.view()

arr2[2]=1000 #will effect the original array

print(arr)

print(arr2)

o/p:

[ 10 20 1000 40 50]

[ 10 20 1000 40 50]

Ex:

To prove the difference between view and copy, we can use “base”. It returns none, if it owns the data i.e. copy. It returns the original object, if it doesn’t own the data i.e. view

import numpy as np

arr=np.array([10,20,30,40,50])

arr2=arr.view()

print(arr.base)

print(arr2.base)

o/p:

None

[10 20 30 40 50]

Ex:

import numpy as np

arr=np.array([10,20,30,40,50])

arr2=arr.copy()

print(arr.base)

print(arr2.base)

o/p:

None

None

Getting the shape of an array

import numpy as np

arr=np.array([[10,20,30,40,50],[100,200,300,400,500]])

print(arr.shape)

o/p: (2,5)

Ex:

import numpy as np

arr=np.array([[1,2,3],[10,20,30]])

print(arr.shape)

print(arr)

o/p:

(2, 3)

[[ 1 2 3]

[10 20 30]]

Reshaping numpy arrays

Ex:

import numpy as np

arr=np.array([[1,2,3],[10,20,30]])

print(arr.shape)

arr2=arr.reshape(3,2)

print(arr2.shape)

o/p:

(2,3)

(3,2)

Ex:

Converting 1D array to 2D

import numpy as np

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

arr2=arr.reshape(2,5)

print(arr2.shape)

print(arr2)

o/p:

(2, 5)

[[ 1 2 3 4 5]

[ 6 7 8 9 10]]

Ex:

Converting from 1D to 3D

import numpy as np

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

arr2=arr.reshape(2,3,2)

print(arr2.shape)

print(arr2)

o/p:

(2, 3, 2)

[[[ 1 2]

[ 3 4]

[ 5 6]]

[[ 7 8]

[ 9 10]

[11 12]]]

Ex:

Flattening the arrays

Flattening array means converting a multidimensional array into a 1D array.

We can use reshape(-1) to do this.

Example

Convert the array into a 1D array:

import numpy as np

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

newarr = arr.reshape(-1)

print(newarr)

o/p: [1 2 3 4 5 6]

Numpy array iteration

1-D array iteration:

import numpy as np

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

for i in arr:

print(i)

o/p:

1

2

3

2-D array iteration:

import numpy as np

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

for i in arr:

print(i)

for i in arr:

for j in i:

print(j)

o/p:

[1 2 3]

[4 5 6]

1

2

3

4

5

6

3-D array iteration:

import numpy as np

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

[[10,20,30],[40,50,60]]])

for i in arr:

for j in i:

for k in j:

print(k)

o/p:

1

2

3

4

5

6

10

20

30

40

50

60

To make iteration simple, we can use np.nditer(arr) method:

To write for loops on n-dimensional arrays, will be tedious. So, we can use nditer()

Ex:

import numpy as np

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

[[10,20,30],[40,50,60]]])

for i in np.nditer(arr):

print(i)

o/p:

1

2

3

4

5

6

10

20

30

40

50

60

Changing the increment range(step size):

import numpy as np

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

[[10,20,30],[40,50,60]]])

for i in np.nditer(arr[:,::2]):

print(i)

o/p:

1

2

3

10

20

30

On a 2-D array,

import numpy as np

arr=np.array([[1,2,3],[10,20,30]])

for i in np.nditer(arr[:,::2]):

print(i)

o/p:

1

3

10

30

Numpy array joining

Concatenate ():

import numpy as np

arr1 = np.array([1, 2, 3])

arr2 = np.array([4, 5, 6])

arr = np.concatenate((arr1, arr2))

print(arr)

o/p:

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

Joining along rows:

import numpy as np

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

arr2 = np.array([[5,6],[7,8]])

arr = np.concatenate((arr1, arr2),axis=1)

print(arr)

o/p:

[[1 2 5 6]

[3 4 7 8]]

Joining along columns:

import numpy as np

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

arr2 = np.array([[5,6],[7,8]])

arr = np.concatenate((arr1, arr2),axis=0)

print(arr)

o/p:

[[1 2]

[3 4]

[5 6]

[7 8]]

Splitting numpy arrays

array-split(array,no.of parts we want to split)

Ex:

import numpy as np

arr1 =np.array([1,2,3,4,5,6,7,8,9])

arr2=np.array\_split(arr1,3)

print(arr2)

o/p:

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

The return value contains 3 arrays

If the array has less elements than required, it will adjust at the end.

Ex:

import numpy as np

arr1 =np.array([1,2,3,4,5,6,7,8,9])

arr2=np.array\_split(arr1,4)

print(arr2)

o/p:

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

To access the splitted arrays,

arr2[0],arr2[1],arr2[2]

Splitting 2-D arrays:

import numpy as np

arr1 =np.array([[1,2,3,4,5],[10,20,30,40,50]])

arr2=np.array\_split(arr1,2)

print(arr2)

o/p:

[1,2,3,4,5],[10,20,30,40,50]

Searching numpy arrays

Where():

import numpy as np

arr =np.array([10,20,30,40,50,60,70,80,90])

ele=np.where(arr==40) #returns the position/index where “40” is located

print(ele)

o/p:

array[3],dtype=int64

What if multiple occurrences of 40 are there in the array:

import numpy as np

arr =np.array([10,20,30,40,50,60,70,80,90,40])

ele=np.where(arr==40)

print(ele)

o/p:

array([3,9],dtype=int64)

Ex: To get the indices where even elements are present in the array

import numpy as np

arr =np.array([10,20,31,43,50,60,70,80,90,40])

ele=np.where(arr%2==0)

print(ele)

o/p:

array([0, 1, 4, 5, 6, 7, 8, 9], dtype=int64),

Ex: To get the indices where odd elements are located,

import numpy as np

arr =np.array([10,20,31,43,50,60,70,80,90,40])

ele=np.where(arr%2==1)

print(ele)

o/p:

array([2,3])

Numpy Sorting

import numpy as np

arr =np.array([10,20,31,43,50,60,70,80,90,40])

print(np.sort(arr))

Ex:

[10 20 31 40 43 50 60 70 80 90]

Original array is unchanged

Filtering arrays:2 ways, 1. Using a list 2. Using array directly

Getting required elements out of the existing array and creating a new array out of them is called “filtering”.

Filtering using Boolean index list:

import numpy as np

arr =np.array([10,20,31,100])

list1=[True,False,True,False]

newarr=arr[list1]

print(newarr)

o/p:

[10 31]

Ex:

Create a new array with elements greater than 50

import numpy as np

arr =np.array([10,20,31,100,89,34,70])

newarr=[]

for i in arr:

if i>50:

newarr.append(i)

print(newarr)

o/p:

[100,89,70]

(or)

import numpy as np

arr =np.array([10,20,31,100,89,34,70])

newarr=[]

for i in arr:

if i>50:

newarr.append(True)

else:

newarr.append(False)

print(arr[newarr])

Ex: Create an array that contains only even elements from the original array

import numpy as np

arr =np.array([10,20,31,100,89,34,70])

newarr=[]

for i in arr:

if i%2==0:

newarr.append(True)

else:

newarr.append(False)

print(arr[newarr])

o/p:

[10 20 100 34 70]

Creating filter directly from array

import numpy as np

arr =np.array([10,20,31,100,89,34,70])

newarr=arr>50

print(newarr)

arr2=arr[newarr]

print(arr2)

o/p:

[False False False True True False True]

[100 89 70]

Ex:

Creating even element arrays

import numpy as np

arr =np.array([10,20,31,100,89,34,70])

newarr=arr%2==0

print(newarr)

arr2=arr[newarr]

print(arr2)

o/p:

[True True False True False True True]

[10 20 100 34 70]

Random numbers in numpy

Numpy offer random module to work with random numbers

Ex:

from numpy import random

print(random.randint(100))

o/p:

23

random.rand(): a random float between 0 and 1

Generate random array:

from numpy import random

arr=random.randint(100,size=5) # 5 random integers between 0 and 100

print(arr)

o/p:

[0 0 90 9 23]

To create a 2-D array with 3 rows, each row containing 5 random integers between 0 to 100

Ex:

from numpy import random

arr=random.randint(100,size=(3,5))

print(arr)

o/p:

[[85 6 75 95 37]

[62 45 74 36 66]

[55 57 67 1 47]]

Ex: create an array containing 5 random floats

from numpy import random

arr=random.rand(5)

print(arr)

o/p:

[0.15964102 0.13473993 0.61349147 0.41660549 0.7945103 ]

Ex: create a 2-D array with 3 rows, each row having 5 random numbers

from numpy import random

arr=random.rand(3,5)

print(arr)

Numpy functions

import numpy as np

list1=[1,2,3]

list2=[10,20,30]

arr=np.add(list1,list2)

print(arr)

o/p:

[11 22 33]