learning numpy

December 28, 2022

0.1 Numpy Notes

by Karthik Nair

Notes compiled from multiple sources including the Official Numpy Documentation

0.1.1 Numpy - introduction

NumPy is the fundamental package for scientific computing in Python. It is a Python library that provides a multi-dimensional array object, various derived objects (such as masked arrays and matrices), and an assortment of routines for fast operations on arrays, including mathematical, logical, shape manipulation, sorting, selecting, I/O, discrete Fourier transforms, basic linear algebra, basic statistical operations, random simulation and much more

0.1.2 Why is numpy faster than lists?

Numpy uses ndarray. An ndarray is a collection of homogeneous data-types that are stored in contiguous memory locations. On the other hand, a list in Python is a collection of heterogeneous data types stored in non-contiguous memory locations. The NumPy package breaks down a task into multiple fragments and then processes all the fragments parallelly.

Ndarry classes are used to represent both Matrices and Vectors

0.1.3 How are NumPy Arrays different from Python lists?

NumPy arrays have a fixed size at creation, unlike Python lists (which can grow dynamically). Changing the size of an ndarray will create a new array and delete the original

The elements in a NumPy array are all required to be of the same data type, and thus will be the same size in memory. The exception: one can have arrays of (Python, including NumPy) objects, thereby allowing for array of different sized elements

The NumPy package integrates C, C++, and Fortran codes in Python.
These programming languages have very little execution time compared to Python

0.1.4 Importing the NumPy library

[1]:
```
import numpy as np # it has become a convention to use np as the alias for␣
↪numpy
```

0.1.5 Creating NumPy Arrays

Using `array` function You can create an array from a regular Python list or tuple using the array function

[2]:
```
my_array=np.array([1,2,3,4]) #from list

new_array=np.array((34,2,0,2)) #from tuple

print(my_array)
print(new_array)
```

```
[1 2 3 4]
[34  2  0  2]
```

An optional parameter `ndmin` specifies the minimum number of dimensions that the resulting array should have. > Ones will be prepended to the shape as needed to meet this requirement.

[3]:
```
new_array=np.array([32,11,43,22],ndmin=3)
print(new_array)
print(new_array.shape)
```

```
[[[32 11 43 22]]]
(1, 1, 4)
```

[4]:
```
array0=np.array([1,2,3,4])

print(type(array0)) # returns <class 'numpy.ndarray'>
```

```
<class 'numpy.ndarray'>
```

Using *placeholder* functions Placeholder functions are used when we need to create an array with its size known, but elements unknown (intially) For example, we might need to create an array to store data of 10 students where the elements would be filled later

np.zeros : creates an array full of zeros

[5]:
```
np.zeros((2,4))
```

[5]:
```
array([[0., 0., 0., 0.],
       [0., 0., 0., 0.]])
```

np.ones : returns an array full of ones

[6]:
```
np.ones((2,4))
```

[6]:
```
array([[1., 1., 1., 1.],
       [1., 1., 1., 1.]])
```

np.empty : returns an array whose initial content is random

[7]:
```
np.empty((2,2)) #values could be anything
```

[7]:
```
array([[4.65317268e-310, 0.00000000e+000],
       [4.65418846e-310, 4.65418852e-310]])
```

np.full : returns an array of the given shape filled with the fill_value

[8]:
```
np.full((2,2), fill_value=10)
np.full((2,2), 10)
```

[8]:
```
array([[10, 10],
       [10, 10]])
```

Functions to create sequence of numbers np.arange : create sequences of numbers in a format analogous to Python's built-in `range`

[9]:
```
a=np.arange(10)
a
```

[9]:
```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

[10]:
```
np.arange( 10, 30, 5 )
```

[10]:
```
array([10, 15, 20, 25])
```

[11]:
```
np.arange( 0, 2, 0.3 ) # it also accepts float arguments
```

[11]:
```
array([0. , 0.3, 0.6, 0.9, 1.2, 1.5, 1.8])
```

np.linspace : receives as an argument the number of elements that we want, instead of the step values

[12]:
```
np.linspace( 0, 2, 9 )
```

[12]:
```
array([0.  , 0.25, 0.5 , 0.75, 1.  , 1.25, 1.5 , 1.75, 2.  ])
```

0.1.6 Dimensions of an array

Ndarrays can be of different dimensions or **Axes** e.g. 0D , 1D (known as Vector), 2D (known as Matrix), 3D, etc Arrays of dimensions 3 or higher are commonly known as **Tensors**

Rank of an Array is the number of dimensions in an array

0.1.7 Shape of an array

Shape of an array is a tuple with

0.1.8 Arrays of different dimensions

0D Arrays Zero dimensional arrays are **scalars**. They can't be accessed using indices but can be converted into scalars Zero dimensional arrays are NOT equivalent to length-1 1D arrays

[13]:
```
# Creating a zero dimensional array

array1=np.array(27)

array1
```

[13]:
```
array(27)
```

[14]:
```
array1.shape # returns () , which indicates zero  dimension
```

[14]:
```
()
```

[15]:
```
array1.ndim # returns 0, which indicates 0 dimension
```

[15]:
```
0
```

[16]:
```
# converting zero-dimensional arrays to scalars
scalar_array=array1.item()
scalar_array
```

[16]:
```
27
```

0.1.9 Note

For a zero dimensional array:
`print(array1[0]) #results in an error`

- Trying to access the value in a 0D array using index results in an error
  reason : 0D ndarrays are considered scalars

1D Arrays In a 1D array, a single argument is passed to access a specific value e.g in array [1,2,3,4], element '3' can be accessed with its index 2

[17]:
```
# Creating a length-1 one dimensional array

array2=np.array([27]) #unlike zero-dimensional arrays, in len1 1D arrays, we␣
 ↪pass the array element in sq brackets

array2
```

[17]:
```
array([27])
```

[18]:
```
# with length more than 1

array3=np.array([1,2,3,4])

array3
```

[18]:
```
array([1, 2, 3, 4])
```

[19]:
```
array2[0] # element can be accessed using index
```

[19]:
```
27
```

[20]:
```
array3[2] # accessing the third element in array
```

[20]:
```
3
```

[21]:
```
array3.shape # returns (4,) which indicates an array with 1 dimension and 4␣
↪elements
```

[21]:
```
(4,)
```

[22]:
```
array3.ndim # returns 1, indicating 1 dimension
```

[22]:
```
1
```

2D Arrays In 2D array, 2 arguments needs to be passed to access a single element

[23]:
```
array4=np.array([[1,2,3],[43,22,11]]) # Creating a 2D numpy array

array4

# for instance, np.array([[1,2,3],[3,4,22],[43,22,11]]) indicates a 2D array␣
↪with 3 layers and 3 elements in each layer
```

[23]:
```
array([[ 1,  2,  3],
       [43, 22, 11]])
```

[24]:
```
array4[1][1]
# Accessing the second element from the 2nd layer of the 2nd dimension
```

[24]:
```
22
```

[25]:
```
array4.shape # returns (2,3)
```

[25]:
```
(2, 3)
```

[26]:
```
array4.ndim
```

[26]: 2

3D Arrays In 3D array, 3 argument needs to be passed to access a single element

[27]:
```
array5=np.array([[[22,3,32],[33,1,32]],[[44,34,21],[3,29,32]]])
# creating a 3D array with 2 layers and 2 sub-layers within each layer and 3 elements in each sub-layer

array5
```

[27]:
```
array([[[22,  3, 32],
        [33,  1, 32]],

       [[44, 34, 21],
        [ 3, 29, 32]]])
```

[28]:
```
array5[1][1][1]
# accessing the 2nd element of the 2nd sub-layer of the second layer, ie 29
```

[28]: 29

[29]:
```
array5.ndim
```

[29]: 3

[30]:
```
array5.shape
```

[30]: (2, 2, 3)

[31]:
```
array6=np.
  ↪array([[[22,3,32],[33,1,32],[33,42,38]],[[44,34,21],[3,22,32],[99,43,88]]])
# creating a 3D array with 2 layers and 3 sub-layers within each layer and 3 elements in each layer

array6
```

[31]:
```
array([[[22,  3, 32],
        [33,  1, 32],
        [33, 42, 38]],

       [[44, 34, 21],
        [ 3, 22, 32],
        [99, 43, 88]]])
```

[32]:
```
array6.ndim
```

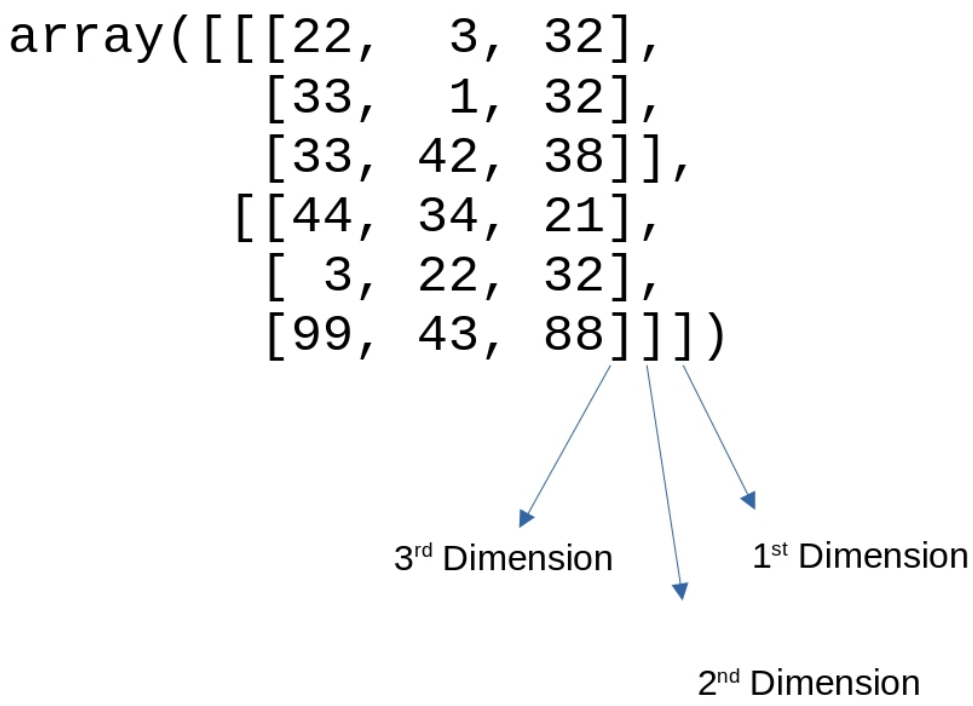
[32]: 3

[33]:
```
array6.shape
```

[33]:
```
(2, 3, 3)
```

0.1.10 Printing Arrays

When you print an array, NumPy displays it in a similar way to nested lists, but with the following layout: • the last axis is printed from left to right, • the second-to-last is printed from top to bottom, • the rest are also printed from top to bottom, with each slice separated from the next by an empty line.



One-dimensional arrays are then printed as rows, bidimensionals as matrices and tridimensionals as lists of matrices

If an array is too large to be printed, NumPy automatically skips the central part of the array and only prints the corners:

To disable this behaviour and force numpy to print the whole array, we can change the printing options by using the np.set_printoptions method and passing either of the two arguments :

- np.set_printoptions(threshold=np.inf)
- np.set_printoptions(threshold=sys.maxsize) after importing sys

[34]:
```
import sys
np.set_printoptions(threshold=sys.maxsize)
# now printing array a would print all the numbers
```

0.1.11 Some important attributes of an Ndarray

`ndarray.ndim` Lets you see the number of axes(dimensions) of a numpy array
`ndarray.shape` Returns a tuple of integers indicating the size of the array in each dimension e.g. for a matrix with n rows and m columns, shape will be (n,m). (len(ndarray.shape)==ndarray.ndim)
`ndarray.size` Displays the total number of elements in an array. (This is equal to the product of elements of shape)
`ndarray.dtype` returns an object describing the type of elements in the array
`ndarray.itemsize` returns size (in bytes) of each element of the array (Equivalent to ndarray.dtype.itemsize)
`ndarray.data` returns the buffer containing the actual elements of the array

0.1.12 Basic Array Operations

Arithmetic operators on arrays apply elementwise

[35]:
```
a = np.array( [20,30,40,50] )
b = np.arange( 4 )
b
```

[35]:
```
array([0, 1, 2, 3])
```

[36]:
```
c=a-b
c
```

[36]:
```
array([20, 29, 38, 47])
```

[37]:
```
b**2
```

[37]:
```
array([0, 1, 4, 9])
```

[38]:
```
10*np.sin(a)
```

[38]:
```
array([ 9.12945251, -9.88031624,  7.4511316 , -2.62374854])
```

[39]:
```
a<30
```

[39]:
```
array([ True, False, False, False])
```

The product operator * operates elementwise in NumPy arrrays

[40]:
```
a2=np.array([[1,2,3],[4,5,6]])
```

[41]:
```
a3=np.array([[7,8,9],[10,11,12]])
```

[42]:
```
a2*a3
```

[42]:
```
array([[ 7, 16, 27],
       [40, 55, 72]])
```

The matrix product can be performed using the @ operator (in python >=3.5) or the dot function or method:

[43]:
```
a4=np.array([[3,4],[2,2]])
```

[44]:
```
a5=np.array([[4,2],[9,4]])
```

[45]:
```
a4@a5 #using @ operator
```

[45]:
```
array([[48, 22],
       [26, 12]])
```

[46]:
```
a4.dot(a5) #using dot function
```

[46]:
```
array([[48, 22],
       [26, 12]])
```

[47]:
```
a6=np.array([[1,2,3],[4,5,6]])
```

[48]:
```
a7=np.array([[7,8],[9,10],[11,12]])
```

[49]:
```
a6@a7 # matrix product of a 2*3 and 3*2 matrix
```

[49]:
```
array([[ 58,  64],
       [139, 154]])
```

[50]:
```
 a6.dot(a7) # also does matrix product
```

[50]:
```
array([[ 58,  64],
       [139, 154]])
```

[ ]:

[ ]:

[ ]:

[ ]: