1-numpy-array-object

March 8, 2023

1 The Numpy array object

2 NumPy Arrays

python objects:

1. high-level number objects: integers, floating point
2. containers: lists (costless insertion and append), dictionaries (fast lookup)

Numpy provides:

1. extension package to Python for multi-dimensional arrays
2. closer to hardware (efficiency)
3. designed for scientific computation (convenience)
4. Also known as array oriented computing

[1]:
```
import numpy as np
a = np.array([0, 1, 2, 3])
print(a)

print(np.arange(10))
```

```
[0 1 2 3]
[0 1 2 3 4 5 6 7 8 9]
```

Why it is useful: Memory-efficient container that provides fast numerical operations.

[ ]:
```
#python lists
L = range(1000)
%timeit [i**2 for i in L]
```

```
307 µs ± 17.6 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

[ ]:
```
a = np.arange(1000)
%timeit a**2
```

```
1.35 µs ± 126 ns per loop (mean ± std. dev. of 7 runs, 1000000 loops each)
```

3 1. Creating arrays → Every 1-D array is a "column" array

** 1.1. Manual Construction of arrays**

[ ]:
```
#1-D

a = np.array([0, 1, 2, 3])

a
```

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

[ ]:
```
#print dimensions

a.ndim
```

[ ]:
```
1
```

[ ]:
```
#shape

a.shape
```

Even if you do the "Transpose" of 'a'. The shape will be the same.

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

→ if you are making only column array, it is not possible because there always be row. [[1]
[2]
[3]]

[ ]:
```
len(a)
```

[ ]:
```
4
```

[ ]:
```
# 2-D, 3-D....

b = np.array([[0, 1, 2], [3, 4, 5]])

b
```

[ ]:
```
array([[0, 1, 2],
       [3, 4, 5]])
```

[ ]:
```
b.ndim
```

[ ]:
```
2
```

[ ]:
```
b.shape
```

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

[ ]:
```
len(b) #returns the size of the first dimention
```

[ ]:
```
2
```

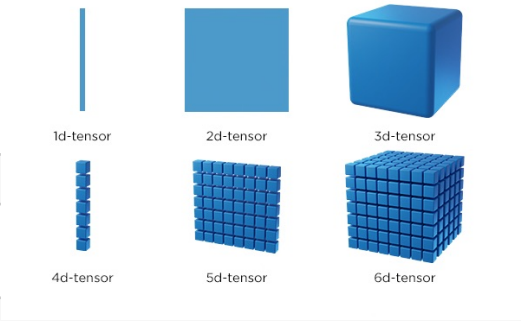
```
[ ]: c = np.array([[[0, 1], [2, 3]], [[4, 5], [6, 7]]])

     c
```

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

            [[4, 5],
             [6, 7]]])
```

→ 1D array always look or pretend as a column array by default. (imp.)




```
[ ]: c.ndim
```

```
[ ]: 3
```

```
[ ]: c.shape
```

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

** 1.2 Functions for creating arrays**

```
[ ]: #using arrange function

     # arange is an array-valued version of the built-in Python range function

     a = np.arange(10) # 0.... n-1
     a
```

→ you can also create 2-D or 3-D or multiple n-D array using arange(), linspace(). But not directly, by using ".reshape" you can create it.

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

```
[ ]: b = np.arange(1, 10, 2) #start, end (exclusive), step

     b
```

```
[ ]: array([1, 3, 5, 7, 9])
```

```
[ ]: #using linspace

     a = np.linspace(0, 1, 6) #start, end, number of points

     a
```

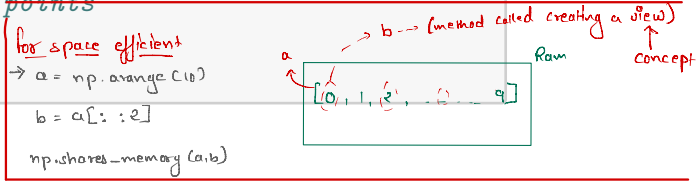
```
[ ]: array([ 0. ,  0.2,  0.4,  0.6,  0.8,  1. ])
```

```
[ ]: #common arrays

     a = np.ones((3, 3))
```

Q. diff. b/w 1D array and list?

→ you can see below 'array' is taking more space. But, here you can see it store data more efficiently with other variables.




```
import numpy as np
import sys
a = np.arange(10)
ab = [0,1,2,3,4,5,6,7,8,9]
print(sys.getsizeof(a))
print(sys.getsizeof(ab))
print(type(a),a)
print(type(ab),ab)
```

```
192
136
<class 'numpy.ndarray'> [0 1 2 3 4 5 6 7 8 9]
<class 'list'> [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

→ you can use '.zeros' and '.ones' to create 3-D array directly eg:- a = np.zeros((2,3,4))
'2' layers of '3', '4' matrix.

```
a
```

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

```
b = np.zeros((3, 3))

b
```

→ you can't use 'eye', 'diag' to create 3-D array directly.

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

```
c = np.eye(3)  #Return a 2-D array with ones on the diagonal and zeros␣
↪elsewhere.

c
```

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

```
d = np.eye(3, 2) #3 is number of rows, 2 is number of columns, index of␣
↪diagonal start with 0

d
```

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

```
#create array using diag function

a = np.diag([1, 2, 3, 4]) #construct a diagonal array.

a
```

```
array([[1, 0, 0, 0],
       [0, 2, 0, 0],
       [0, 0, 3, 0],
       [0, 0, 0, 4]])
```

```
np.diag(a)   #Extract diagonal
```

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

→ you can use .random to create N-Dimension array.

[ ]:
```
#create array using random

#Create an array of the given shape and populate it with random samples from a␣
 ↪uniform distribution over [0, 1).
a = np.random.rand(4)

a
```

[ ]:
```
array([ 0.85434586,  0.05106692,  0.37337949,  0.32093548])
```

[ ]:
```
a = np.random.randn(4)#Return a sample (or samples) from the "standard normal"␣
 ↪distribution.  ***Gausian***

a
```

[ ]:
```
array([  1.99407539e+00,  -1.33836224e+00,   3.07395038e-04,
         4.73482900e-01])
```

Note:

For random samples from N(, ^2), use:

sigma * np.random.randn(...) + mu

4 2. Basic DataTypes

You may have noticed that, in some instances, array elements are displayed with a **trailing dot (e.g. 2. vs 2)**. This is due to a difference in the **data-type** used:

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

a.dtype
```

[ ]:
```
dtype('int64')
```

[ ]:
```
#You can explicitly specify which data-type you want:

a = np.arange(10, dtype='float64')
a
```

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

[ ]:
```
#The default data type is float for zeros and ones function

a = np.zeros((3, 3))

print(a)
```

```
a.dtype
```

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

[ ]:
```
dtype('float64')
```

other datatypes

[ ]:
```
d = np.array([1+2j, 2+4j])   #Complex datatype

print(d.dtype)
```

```
complex128
```

[ ]:
```
b = np.array([True, False, True, False])  #Boolean datatype

print(b.dtype)
```

```
bool
```

[2]:
```
s = np.array(['Ram', 'Robert', 'Rahim'])

s.dtype
```

[2]:
```
dtype('<U6')
```

Each built-in data type has a character code that uniquely identifies it.

'b' – boolean

'i' – (signed) integer

'u' – unsigned integer

'f' – floating-point

'c' – complex-floating point

'm' – timedelta

'M' – datetime

'O' – (Python) objects

'S', 'a' – (byte-)string

'U' – Unicode

'V' – raw data (void)

For more details

https://docs.scipy.org/doc/numpy-1.10.1/user/basics.types.html

5 3. Indexing and Slicing

3.1 Indexing → How to call elements from "array". and you can also replace it.

The items of an array can be accessed and assigned to the same way as other **Python sequences (e.g. lists)**:

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

print(a[5])  #indices begin at 0, like other Python sequences (and C/C++)
```

5

[ ]:
```
# For multidimensional arrays, indexes are tuples of integers:

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

print(a[2, 2])
```

3

[ ]:
```
a[2, 1] = 5 #assigning value

a
```

[ ]:
```
array([[1, 0, 0],
       [0, 2, 0],
       [0, 5, 3]])
```

3.2 Slicing → you can use that in N-Dim. array too.

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

a
```

→ arr = np.diag([1,2,3])
print(arr[0:3,2])
→ [0, 0, 3]

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

[ ]:
```
a[1:8:2] # [startindex: endindex(exclusive) : step]
```

[ ]:
```
array([1, 3, 5, 7])
```

[ ]:
```
#we can also combine assignment and slicing:

a = np.arange(10)
a[5:] = 10
a
```

```
import numpy as np
a = np.random.rand(3,4,3,4)
print(a[:,1:3,0:2,:])
```

[ ]:
```
array([ 0,  1,  2,  3,  4, 10, 10, 10, 10, 10])
```

Note :- a[ ] → index
→ value of 'a' at that index.

a[,] ↔ a[, ,]
row, column ↔ row, Layer, column

[ ]:
```
b = np.arange(5)
a[5:] = b[::-1]  #assigning

a
```

[ ]:
```
array([0, 1, 2, 3, 4, 4, 3, 2, 1, 0])
```

6 4. Copies and Views

A slicing operation creates a view on the original array, which is just a way of accessing array data. Thus the original array is not copied in memory. You can use **np.may_share_memory()** to check if two arrays share the same memory block.

When modifying the view, the original array is modified as well:

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

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

[ ]:
```
b = a[::2]
b
```

[ ]:
```
array([0, 2, 4, 6, 8])
```

[ ]:
```
np.shares_memory(a, b)
```

[ ]:
```
True
```

[ ]:
```
b[0] = 10
b
```

[ ]:
```
array([10,  2,  4,  6,  8])
```

[ ]:
```
a  #eventhough we modified b,  it updated 'a' because both shares same memory
```

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

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

c = a[::2].copy()     #force a copy
c
```

[ ]:
```
array([0, 2, 4, 6, 8])
```

[ ]:
```
np.shares_memory(a, c)
```

[ ]:
```
False
```

[ ]:
```
c[0] = 10

a
```

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

7 5. Fancy Indexing {double index}

NumPy arrays can be indexed with slices, but also with boolean or integer arrays **(masks)**. This method is called **fancy indexing**. It creates copies not views.

Using Boolean Mask

[ ]:
```
a = np.random.randint(0, 20, 15)
a
```

[ ]:
```
array([18, 17,  1, 18,  5, 17,  0, 14, 12, 11,  4, 15, 16,  8,  7])
```

boolean array.

[ ]:
```
mask = (a % 2 == 0)
```
→ it will create a 'boolean mask array'
↳ mask = [T, F, F, T, F, F, T ---- F]

[ ]:
```
extract_from_a = a[mask]

extract_from_a
```
↳ it will print all the value which is 'True'.
→ we can use this in N-Dim. array too.
↳ but you'll only get 1-D array as a resultant.

[ ]:
```
array([18, 18,  0, 14, 12,  4, 16,  8])
```

Indexing with a mask can be very useful to assign a new value to a sub-array:

[ ]:
```
a[mask] = -1
a
```

[ ]:
```
array([-1, 17,  1, -1,  5, 17, -1, -1, -1, 11, -1, 15, -1, -1,  7])
```

Indexing with an array of integers

[ ]:
```
a = np.arange(0, 100, 10)

a
```

[ ]:
```
array([ 0, 10, 20, 30, 40, 50, 60, 70, 80, 90])
```

[ ]:
```
#Indexing can be done with an array of integers, where the same index is␣
↪repeated several time:

a[[2, 3, 2, 4, 2]]
```

→ a[1:5] = array([10, 20, 30, 40])
→ a[[6,7,8]] = array([60, 70, 80])
Only difference is in one square bracket.

→ a = np.diag([1,2,3,4])
mask = (a%2 == 0)
b = a[mask]
b.dtype → (int64)
b.shape → (14,)
b.ndim → 1

```
[ ]: array([20, 30, 20, 40, 20])
```

```
[ ]: # New values can be assigned

     a[[9, 7]] = -200

     a
```

```
[ ]: array([   0,   10,   20,   30,   40,   50,   60, -200,   80, -200])
```

→ you can use that in 2-D array too.

→ `ar = np.diag([1,2,3,4])` ⟶

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 4 \end{bmatrix} \begin{matrix} \leftarrow 0 \\ \leftarrow 1 \\ \leftarrow 2 \\ \leftarrow 3 \end{matrix}$$

`br = ar[[3, 1, 0]]`

↳ br =

$$\begin{bmatrix} 0 & 0 & 0 & 4 \\ 0 & 2 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}$$

↳ it will get more confusing when you do that `[[ ],[ ]]`

So, don't do that.

To study :- How list and array store in memory location or other data structures (tuples, string, Dict,) etc ?