The Numpy array object

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

In [1]:
```
import numpy as np
a = np.array([0, 1, 2, 3])
print(a)
print(type(a))
print(np.arange(10))
```

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

In [2]:
```
b=np.array([1,3,4,'hello'])
print(b)
```

```
['1' '3' '4' 'hello']
```

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

In [ ]:
```
#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)
```

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

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

1. Creating arrays

** 1.1. Manual Construction of arrays**

In [ ]:
```
#1-D

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

a
```

Out[9]:
```
array([0, 1, 2, 3])
```

In [ ]:
```
#print dimensions

a.ndim
```

Out[10]:
```
1
```

In [ ]:
```
#shape

a.shape
```

Out[11]:
```
(4,)
```

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

Out[12]:
```
4
```

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

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

b
```

Out[13]:
```
array([[0, 1, 2],
       [3, 4, 5]])
```

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

Out[14]:
```
2
```

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

Out[15]:
```
(2, 3)
```

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

Out[16]:
```
2
```

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

c
```

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

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

In [6]:
```
print(c.ndim)
print(c.shape)
print(len(c))
```

```
3
(2, 2, 2)
2
```

In [12]:
```
d= np.array([[[2,3]],[[4,5]],[[6,7]]])
d.shape
```

Out[12]:
```
(3, 1, 2)
```

In [4]:
```
import numpy as np
s= np.array([[[0],[1]],[[2],[3]],[[4],[5]]])
s
```

Out[4]:
```
array([[[0],
        [1]],

       [[2],
        [3]],

       [[4],
        [5]]])
```

In [3]:
```
s.shape
```

Out[3]:
```
(3, 2, 1)
```

In [5]:
```
c.ndim
```

Out[5]:
```
3
```

In [8]:
```
len(c)
```

Out[8]:
```
2
```

In [6]:
```
c.shape
```

Out[6]:
```
(2, 2, 2)
```

** 1.2 Functions for creating arrays**

In [ ]:
```
#using arrange function

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

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

Out[17]:
```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

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

b
```

Out[18]:
```
array([1, 3, 5, 7, 9])
```

In [ ]:
```
#using linspace

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

a
```

Out[19]:
```
array([ 0. ,  0.2,  0.4,  0.6,  0.8,  1. ])
```

In [2]:
```
#common arrays

a = np.ones((3, 3)) #row, coloum

a
```

Out[2]:
```
array([[1., 1., 1.],
       [1., 1., 1.],
       [1., 1., 1.]])
```

In [3]:
```


k=1
for i in range(3):
    for j in range(3):
        a[i][j]= k
        k=k+1
print(a)
```

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

In [7]:
```
a[1][1]
```
Out[7]:
```
5.0
```

In [4]:
```
b = np.zeros((3, 3),dtype=int)

b
```
Out[4]:
```
array([[0, 0, 0],
       [0, 0, 0],
       [0, 0, 0]])
```

In [ ]:
```
c = np.eye(3)  #Return a 2-D array with ones on the diagonal and zeros elsewhere.
#identity matrix
c
```
Out[23]:
```
array([[ 1.,  0.,  0.],
       [ 0.,  1.,  0.],
       [ 0.,  0.,  1.]])
```

In [ ]:
```
d = np.eye(3, 2) #3 is number of rows, 2 is number of columns, index of diagonal

d
```
Out[24]:
```
array([[ 1.,  0.],
       [ 0.,  1.],
       [ 0.,  0.]])
```

In [6]:
```
#create array using diag function

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

a
```
Out[6]:
```
array([[1, 0, 0, 0],
       [0, 2, 0, 0],
       [0, 0, 3, 0],
       [0, 0, 0, 4]])
```

In [ ]:
```
np.diag(a)   #Extract diagonal
```
Out[23]:
```
array([1, 2, 3, 4])
```

In [ ]:
```
#create array using random

#Create an array of the given shape and populate it with random samples from a ur
a = np.random.rand(4) #random.random

a
```
Out[24]:
```
array([ 0.85434586,  0.05106692,  0.37337949,  0.32093548])
```

In [ ]:
```
a = np.random.randn(4)#Return a sample (or samples) from the “standard normal” di

a
```

Out[26]:
```
array([  1.99407539e+00,  -1.33836224e+00,   3.07395038e-04,
         4.73482900e-01])
```

Note:

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

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

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:

In [4]:
```
a = np.arange(10)

a.dtype
```

Out[4]:
```
dtype('int32')
```

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

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

Out[27]:
```
array([ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9.])
```

In [ ]:
```
#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.]]
```

Out[28]:
```
dtype('float64')
```

other datatypes

In [3]:
```
d = np.array([1+2j, 2+4j])   #Complex datatype

print(d.dtype)
```

```
complex128
```

In [4]:
```
b = np.array([True, False, True, False])  #Boolean datatype

print(b.dtype)
```

```
bool
```

In [5]:
```
s = np.array(['Ram', 'Robert', 'Rahim'])

s.dtype
```

Out[5]:
```
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 (https://docs.scipy.org/doc/numpy-1.10.1/user/basics.types.html)](https://docs.scipy.org/doc/numpy-1.10.1/user/basics.types.html)

3. Indexing and Slicing

3.1 Indexing

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

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

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

```
5
```

In [5]:
```
# For multidimensional arrays, indexes are tuples of integers:

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

```
[[1 0 0]
 [0 2 0]
 [0 0 3]]
3
```

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

a
```

Out[35]:
```
array([[1, 0, 0],
       [0, 2, 0],
       [0, 5, 3]])
```

3.2 Slicing

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

a
```

Out[36]:
```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

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

Out[37]:
```
array([1, 3, 5, 7])
```

In [7]:
```
#we can also combine assignment and slicing:

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

Out[7]:
```
array([ 0,  1,  2,  3,  4, 10, 10, 10, 10, 10])
```

In [8]:
```
b = np.arange(5) #0,1,2,3,4
print(a[5:])
print(b[::-1])
a[5:] = b[::-1]  #assigning #4,3,2,1,0

a
```

```
[10 10 10 10 10]
[4 3 2 1 0]
```

Out[8]:
```
array([0, 1, 2, 3, 4, 4, 3, 2, 1, 0])
```

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:

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

Out[41]:
```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

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

Out[42]:
```
array([0, 2, 4, 6, 8])
```

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

Out[43]:
```
True
```

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

Out[44]:
```
array([10,  2,  4,  6,  8])
```

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

Out[45]:
```
array([10,  1,  2,  3,  4,  5,  6,  7,  8,  9])
```

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

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

Out[46]:
```
array([0, 2, 4, 6, 8])
```

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

Out[47]:
```
False
```

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

a
```

Out[48]:
```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

5. Fancy Indexing

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

In [8]:
```
a = np.random.randint(0, 20, 15)
a
```

Out[8]:
```
array([10, 17, 10,  3, 16, 10,  1, 10, 18,  4,  9,  0, 12,  9, 16])
```

In [9]:
```
mask = (a % 2 == 0)
```

In [10]:
```
extract_from_a = a[mask]

extract_from_a
```

Out[10]:
```
array([10, 10, 16, 10, 10, 18,  4,  0, 12, 16])
```

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

In [11]:
```
a[mask] = -1
a
```

Out[11]:
```
array([-1, 17, -1,  3, -1, -1,  1, -1, -1, -1,  9, -1, -1,  9, -1])
```

Indexing with an array of integers

In [3]:
```
a = np.arange(0, 100, 10)

a
```

Out[3]:
```
array([ 0, 10, 20, 30, 40, 50, 60, 70, 80, 90])
```

In [ ]:
```
#Indexing can be done with an array of integers, where the same index is repeatec

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

Out[54]:
```
array([20, 30, 20, 40, 20])
```

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

a[[9, 7]] = -200

a
```

Out[55]:
```
array([   0,   10,   20,   30,   40,   50,   60, -200,   80, -200])
```