NumPy

It also has functions for working in domain of linear algebra, fourier transform, and matrices.

NumPy was created in 2005 by Travis Oliphant. It is an open source project and you can use it freely.

NumPy stands for Numerical Python.

Why Use NumPy?

In Python we have lists that serve the purpose of arrays, but they are slow to process.

NumPy aims to provide an array object that is up to 50x faster than traditional Python lists.

The array object in NumPy is called ndarray, it provides a lot of supporting functions that make working with ndarray very easy.

Arrays are very frequently used in data science, where speed and resources are very important.

Create a NumPy ndarray Object

NumPy is used to work with arrays. The array object in NumPy is called ndarray.

We can create a NumPy ndarray object by using the array() function.

In [1]:
```
import numpy as np
```

In [2]:
```
arr=np.array([1,2,4,5,6])
print(arr)
print(type(arr))
```

```
[1 2 4 5 6]
<class 'numpy.ndarray'>
```

In [3]:
```
#To create an ndarray, we can pass a list, tuple or any array-like object into the array() method, and it will be converted into an ndarray:
arr=np.array((1,2,3,4,5,6,7))
print(arr)
print(type(arr))
```

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

Dimensions in Arrays

In [4]:
```
#0-D array
np.array(50)
```

Out[4]:
```
array(50)
```

In [5]:
```
#1-D array
np.array([1,2,4,5,6])
```

Out[5]:
```
array([1, 2, 4, 5, 6])
```

In [6]:
```
#2-D array
np.array([[1,2,3,4],[4,5,6,8]])
```

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

In [7]:
```
#3-D array
np.array([[[1,2,4],[4,5,6],[4,6,9]]])
```

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

Check Number of Dimensions?

In [8]:
```
a=np.array(50)
b=np.array([1,2,3,4,5,6])
c=np.array([[1,2,3,4],[4,5,6,8]])
d=np.array([[[1,2,4],[4,5,6],[4,6,9]]])

print(a.ndim)
print(b.ndim)
print(c.ndim)
print(d.ndim)
```

```
0
1
2
3
```

In [9]:
```
arr1=np.array([4,5,7,6],ndmin=5)
print(arr1)
print('number of array:',arr1.ndim)
```

```
[[[[[4 5 7 6]]]]]
number of array: 5
```

Slicing arrays

Slicing in python means taking elements from one given index to another given index.

We pass slice instead of index like this: [start:end].

We can also define the step, like this: [start:end:step].

If we don't pass start its considered 0

If we don't pass end its considered length of array in that dimension

If we don't pass step its considered 1

In [10]:
```
arr = np.array([1, 2, 3, 4, 5, 6, 7])
print(arr)
```

```
[1 2 3 4 5 6 7]
```

In [11]:
```
arr[1:5]
```

Out[11]:
```
array([2, 3, 4, 5])
```

In [12]:
```
arr[:]
```

Out[12]:
```
array([1, 2, 3, 4, 5, 6, 7])
```

In [13]:
```
arr[4:]
```

Out[13]:
```
array([5, 6, 7])
```

In [14]:
```
arr[:3]
```

Out[14]:
```
array([1, 2, 3])
```

In [15]:
```
arr[1:7:2] #step
```

Out[15]:
```
array([2, 4, 6])
```

In [16]:
```
arr[::2]
```

Out[16]:
```
array([1, 3, 5, 7])
```

Data Types in NumPy

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)

In [17]:
```
arr = np.array(['apple', 'banana', 'cherry'])
type(arr)
```

Out[17]:
```
numpy.ndarray
```

Creating Arrays With a Defined Data Type

We use the array() function to create arrays, this function can take an optional argument: dtype that allows us to define the

In [18]:
```
arr = np.array([1, 2, 3, 4], dtype='S')
print(arr)
print(arr.dtype)

arr=np.array([1,2,3,4],dtype='i4')
print(arr)
print(arr.dtype)
```

```
[b'1' b'2' b'3' b'4']
|S1
[1 2 3 4]
int32
```

For i, u, S and U we can define size as well.

In [19]:
```
arr = np.array([1, 2, 3, 4], dtype='i4')
print(arr)
print(arr.dtype)
```

```
[1 2 3 4]
int32
```

The Difference Between Copy and View

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.

In [20]:
```
arr=np.array([1,2,3,4,5])
x=arr.copy()
arr[0]=42
print(arr)
print(x)
```

```
[42  2  3  4  5]
[1 2 3 4 5]
```

In [21]:
```
arr=np.array([1,2,3,4,5])
x=arr.view()
arr[0]=100
print(x)
print(arr)
```

```
[ 1 100   3   4   5]
[ 1 100   3   4   5]
```

Check if Array Owns it's Data

As mentioned above, copies owns the data, and views does not own the data, but how can we check this?

Every NumPy array has the attribute 'base' that returns 'None' if the array owns the data.

Otherwise, the base attribute refers to the original object.

In [22]:
```
arr=np.array([1,2,3,4,5,6])
x=arr.copy()
y=arr.view()
```

In [23]:
```
print(x.base)
```

```
None
```

In [24]:
```
print(y.base)
```

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

Shape of an Array

The shape of an array is the number of elements in each dimension.

In [25]:
```
arr=np.array([1,2,4,5,6,7])
arr.shape
```

Out[25]:
```
(6,)
```

In [26]:
```
arr = np.array([1, 2, 3, 4], ndmin=5)

print(arr)
print('shape of array :', arr.shape)
```

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

Reshaping arrays

Reshaping means changing the shape of an array.

The shape of an array is the number of elements in each dimension.

By reshaping we can add or remove dimensions or change number of elements in each dimension.

Reshape From 1-D to 2-D

In [27]:
```
arr=np.array([1,2,3,4,5,6,7,8,9])
arr.reshape(3,3)
```

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

In [28]:
```
arr=np.array([1,2,3,4,5,6,7,8,9,11,44,55])
arr.reshape(4,3)
```

Out[28]:
```
array([[ 1,  2,  3],
       [ 4,  5,  6],
       [ 7,  8,  9],
       [11, 44, 55]])
```

In [29]:
```
arr=np.array([1,2,3,4,5,6,7,8])
(arr.reshape(2,2,2).base)
```

Out[29]:
```
array([1, 2, 3, 4, 5, 6, 7, 8])
```

In [30]:
```
import numpy as np

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

print(arr.reshape(2, 4).base)
```

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

In [31]:
```
arr = np.array([1, 2, 3, 4, 5, 6, 7, 8])
newarr = arr.reshape(2, 2, -1)
print(newarr)
```

```
[[[1 2]
  [3 4]]

 [[5 6]
  [7 8]]]
```

In [32]:
```
arr=np.array([1,2,3,4,5,6,7,8,9,11,44,55])
arr.reshape(3,2,-1)
```

Out[32]:
```
array([[[ 1],
        [ 2]],

       [[ 3],
        [ 4]],

       [[ 5],
        [ 6]],

       [[ 7],
        [ 8]],

       [[ 9],
        [11]],

       [[44],
        [55]]])
```

Flattening the arrays

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

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

In [33]:
```
arr = np.array([[1, 2, 3], [4, 5, 6]])

newarr = arr.reshape(-1)

print(newarr)
```

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

Iterating Arrays

Iterating means going through elements one by one.

As we deal with multi-dimensional arrays in numpy, we can do this using basic for loop of python.

If we iterate on a 1-D array it will go through each element one by one.

In [34]:
```
#iterate on the elements of the following 1-D array:
arr=np.array([2,4,6,8,10,12])
for i in arr:
    print(i)
```

```
2
4
6
8
10
12
```

In [35]:
```
#iterating 2-D Arrays
arr=np.array([[1,2,3,4],[2,4,6,8]])
for i in arr:
    for x in x:
        print(x)
```

```
1
2
3
4
2
4
6
8
```

If we iterate on a n-D array it will go through n-1th dimension one by one.

In [36]:
```
arr = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]])
for x in arr:
    for y in x:
        print(y)
```

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

Iterating Arrays Using nditer()

The function "nditer()" is a helping function that can be used from very basic to very advanced iterations. It solves some basic issues which we face in iteration, lets go through it with examples.

Iterating on Each Scalar Element

In basic for loops, iterating through each scalar of an array we need to use n for loops which can be difficult to write for arrays with very high dimensionality.

In [37]:
```
arr = np.array([[1, 2], [3, 4]], dtype='S')

for x in np.nditer(arr):
    print(x)
```

```
b'1'
b'2'
b'3'
b'4'
```

Iterating Array With Different Data Types

We can use op_dtypes argument and pass it the expected datatype to change the datatype of elements while iterating.

NumPy does not change the data type of the element in-place (where the element is in array) so it needs some other space to perform this action, that extra space is called buffer, and in order to enable it in nditer() we pass flags=['buffered'].

In [38]:
```
arr = np.array([1, 2, 3])

for x in np.nditer(arr, flags=['buffered'], op_dtypes=['S']):
  print(x)
```

```
b'1'
b'2'
b'3'
```

In [39]:
```
#iterate through every scalar element of the 2D array skipping 1 element:


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

for x in np.nditer(arr[:, ::2]):
  print(x)
```

```
1
3
5
7
```

In [40]:
```
arr = np.array([[1, 2, 3, 4], [5, 6, 7, 8]])

for idx, x in np.ndenumerate(arr):
  print(idx, x)
```

```
(0, 0) 1
(0, 1) 2
(0, 2) 3
(0, 3) 4
(1, 0) 5
(1, 1) 6
(1, 2) 7
(1, 3) 8
```

In [ ]:
```

```