What is NumPy?

NumPy is a Python library used for working with arrays.

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.

**Data Science:** is a branch of computer science where we study how to store, use and analyze data for deriving information from it.

Why is NumPy Faster Than Lists?

NumPy arrays are stored at one continuous place in memory unlike lists, so processes can access and manipulate them very efficiently.

This behavior is called locality of reference in computer science.

This is the main reason why NumPy is faster than lists. Also it is optimized to work with latest CPU architectures.

Which Language is NumPy written in?

NumPy is a Python library and is written partially in Python, but most of the parts that require fast computation are written in C or C++.

## 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.

### **Example**

import numpy as np  
  
arr = np.array([1, 2, 3, 4, 5])  
  
print(arr)  
  
print(type(arr))

**type():** This built-in Python function tells us the type of the object passed to it. Like in above code it shows that arr is numpy.ndarray type.

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:

### **Example**

Use a tuple to create a NumPy array:

import numpy as np  
  
arr = np.array((1, 2, 3, 4, 5))  
  
print(arr)

## Dimensions in Arrays

A dimension in arrays is one level of array depth (nested arrays).

**nested array:** are arrays that have arrays as their elements.

## 0-D Arrays

0-D arrays, or Scalars, are the elements in an array. Each value in an array is a 0-D array.

### **Example**

Create a 0-D array with value 42

import numpy as np  
  
arr = np.array(42)  
  
print(arr)

## 1-D Arrays

An array that has 0-D arrays as its elements is called uni-dimensional or 1-D array.

These are the most common and basic arrays.

### **Example**

Create a 1-D array containing the values 1,2,3,4,5:

import numpy as np  
  
arr = np.array([1, 2, 3, 4, 5])  
  
print(arr)

## 2-D Arrays

An array that has 1-D arrays as its elements is called a 2-D array.

These are often used to represent matrix or 2nd order tensors.

NumPy has a whole sub module dedicated towards matrix operations called numpy.mat

### **Example**

Create a 2-D array containing two arrays with the values 1,2,3 and 4,5,6:

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

## 3-D arrays

An array that has 2-D arrays (matrices) as its elements is called 3-D array.

These are often used to represent a 3rd order tensor.

### **Example**

Create a 3-D array with two 2-D arrays, both containing two arrays with the values 1,2,3 and 4,5,6:

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

## Check Number of Dimensions?

NumPy Arrays provides the ndim attribute that returns an integer that tells us how many dimensions the array have.

### **Example**

Check how many dimensions the arrays have:

import numpy as np  
  
a = np.array(42)  
b = np.array([1, 2, 3, 4, 5])  
c = np.array([[1, 2, 3], [4, 5, 6]])  
d = np.array([[[1, 2, 3], [4, 5, 6]], [[1, 2, 3], [4, 5, 6]]])  
  
print(a.ndim)  
print(b.ndim)  
print(c.ndim)  
print(d.ndim)

## Higher Dimensional Arrays

An array can have any number of dimensions.

When the array is created, you can define the number of dimensions by using the ndmin argument.

### **Example**

Create an array with 5 dimensions and verify that it has 5 dimensions:

import numpy as np  
  
arr = np.array([1, 2, 3, 4], ndmin=5)  
  
print(arr)  
print('number of dimensions :', arr.ndim)

In this array the innermost dimension (5th dim) has 4 elements, the 4th dim has 1 element that is the vector, the 3rd dim has 1 element that is the matrix with the vector, the 2nd dim has 1 element that is 3D array and 1st dim has 1 element that is a 4D array.

# **NumPy Array Indexing**

## Access Array Elements

Array indexing is the same as accessing an array element.

You can access an array element by referring to its index number.

The indexes in NumPy arrays start with 0, meaning that the first element has index 0, and the second has index 1 etc.

### **Example**

Get the first element from the following array:

import numpy as np  
  
arr = np.array([1, 2, 3, 4])  
  
print(arr[0])

### **Example**

Get the second element from the following array.

import numpy as np  
  
arr = np.array([1, 2, 3, 4])  
  
print(arr[1])

### **Example**

Get third and fourth elements from the following array and add them.

import numpy as np  
  
arr = np.array([1, 2, 3, 4])  
  
print(arr[2] + arr[3])

## Access 2-D Arrays

To access elements from 2-D arrays we can use comma separated integers representing the dimension and the index of the element.

Think of 2-D arrays like a table with rows and columns, where the dimension represents the row and the index represents the column.

### **Example**

Access the element on the first row, second column:

import numpy as np  
  
arr = np.array([[1,2,3,4,5], [6,7,8,9,10]])  
  
print('2nd element on 1st row: ', arr[0, 1])

### **Example**

Access the element on the 2nd row, 5th column:

import numpy as np  
  
arr = np.array([[1,2,3,4,5], [6,7,8,9,10]])  
  
print('5th element on 2nd row: ', arr[1, 4])

## Access 3-D Arrays

To access elements from 3-D arrays we can use comma separated integers representing the dimensions and the index of the element.

### **Example**

Access the third element of the second array of the first array:

import numpy as np  
  
arr = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]])  
  
print(arr[0, 1, 2])

### **Example Explained**

arr[0, 1, 2] prints the value 6.

And this is why:

The first number represents the first dimension, which contains two arrays:  
[[1, 2, 3], [4, 5, 6]]  
and:  
[[7, 8, 9], [10, 11, 12]]  
Since we selected 0, we are left with the first array:  
[[1, 2, 3], [4, 5, 6]]

The second number represents the second dimension, which also contains two arrays:  
[1, 2, 3]  
and:  
[4, 5, 6]  
Since we selected 1, we are left with the second array:  
[4, 5, 6]

The third number represents the third dimension, which contains three values:  
4  
5  
6  
Since we selected 2, we end up with the third value:  
6

## Negative Indexing

Use negative indexing to access an array from the end.

### **Example**

Print the last element from the 2nd dim:

import numpy as np  
  
arr = np.array([[1,2,3,4,5], [6,7,8,9,10]])  
  
print('Last element from 2nd dim: ', arr[1, -1])

# **NumPy Array Slicing**

## 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

### **Example**

Slice elements from index 1 to index 5 from the following array:

import numpy as np  
  
arr = np.array([1, 2, 3, 4, 5, 6, 7])  
  
print(arr[1:5])

**Note:** The result includes the start index, but excludes the end index.

### **Example**

Slice elements from index 4 to the end of the array:

import numpy as np  
  
arr = np.array([1, 2, 3, 4, 5, 6, 7])  
  
print(arr[4:])

### **Example**

Slice elements from the beginning to index 4 (not included):

import numpy as np  
  
arr = np.array([1, 2, 3, 4, 5, 6, 7])  
  
print(arr[:4])

## Negative Slicing

Use the minus operator to refer to an index from the end:

### **Example**

Slice from the index 3 from the end to index 1 from the end:

import numpy as np  
  
arr = np.array([1, 2, 3, 4, 5, 6, 7])  
  
print(arr[-3:-1])

## STEP

Use the step value to determine the step of the slicing:

### **Example**

Return every other element from index 1 to index 5:

import numpy as np  
  
arr = np.array([1, 2, 3, 4, 5, 6, 7])  
  
print(arr[1:5:2])

### **Example**

Return every other element from the entire array:

import numpy as np  
  
arr = np.array([1, 2, 3, 4, 5, 6, 7])  
  
print(arr[::2])

## Slicing 2-D Arrays

### **Example**

From the second element, slice elements from index 1 to index 4 (not included):

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

**Note:** Remember that second element has index 1.

### **Example**

From both elements, return index 2:

import numpy as np  
  
arr = np.array([[1, 2, 3, 4, 5], [6, 7, 8, 9, 10]])  
  
print(arr[0:2, 2])

### **Example**

From both elements, slice index 1 to index 4 (not included), this will return a 2-D array:

import numpy as np  
  
arr = np.array([[1, 2, 3, 4, 5], [6, 7, 8, 9, 10]])  
  
print(arr[0:2, 1:4])

# **NumPy Array Shape**

## Shape of an Array

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

## Get the Shape of an Array

NumPy arrays have an attribute called shape that returns a tuple with each index having the number of corresponding elements.

### **Example**

Print the shape of a 2-D array:

import numpy as np  
  
arr = np.array([[1, 2, 3, 4], [5, 6, 7, 8]])  
  
print(arr.shape)

The example above returns (2, 4), which means that the array has 2 dimensions, where the first dimension has 2 elements and the second has 4.

### **Example**

Create an array with 5 dimensions using ndmin using a vector with values 1,2,3,4 and verify that last dimension has value 4:

import numpy as np  
  
arr = np.array([1, 2, 3, 4], ndmin=5)  
  
print(arr)  
print('shape of array :', arr.shape)

## What does the shape tuple represent?

Integers at every index tells about the number of elements the corresponding dimension has.

In the example above at index-4 we have value 4, so we can say that 5th ( 4 + 1 th) dimension has 4 elements.

# **NumPy Array Reshaping**

## 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

### **Example**

Convert the following 1-D array with 12 elements into a 2-D array.

The outermost dimension will have 4 arrays, each with 3 elements:

import numpy as np  
  
arr = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12])  
  
newarr = arr.reshape(4, 3)  
  
print(newarr)

## Reshape From 1-D to 3-D

### **Example**

Convert the following 1-D array with 12 elements into a 3-D array.

The outermost dimension will have 2 arrays that contains 3 arrays, each with 2 elements:

import numpy as np  
  
arr = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12])  
  
newarr = arr.reshape(2, 3, 2)  
  
print(newarr)

## Can We Reshape Into any Shape?

Yes, as long as the elements required for reshaping are equal in both shapes.

We can reshape an 8 elements 1D array into 4 elements in 2 rows 2D array but we cannot reshape it into a 3 elements 3 rows 2D array as that would require 3x3 = 9 elements.

### **Example**

Try converting 1D array with 8 elements to a 2D array with 3 elements in each dimension (will raise an error):

import numpy as np  
  
arr = np.array([1, 2, 3, 4, 5, 6, 7, 8])  
  
newarr = arr.reshape(3, 3)  
  
print(newarr)

## Returns Copy or View?

### **Example**

Check if the returned array is a copy or a view:

import numpy as np  
  
arr = np.array([1, 2, 3, 4, 5, 6, 7, 8])  
  
print(arr.reshape(2, 4).base)

The example above returns the original array, so it is a view.

## Unknown Dimension

You are allowed to have one "unknown" dimension.

Meaning that you do not have to specify an exact number for one of the dimensions in the reshape method.

Pass -1 as the value, and NumPy will calculate this number for you.

### **Example**

Convert 1D array with 8 elements to 3D array with 2x2 elements:

import numpy as np  
  
arr = np.array([1, 2, 3, 4, 5, 6, 7, 8])  
  
newarr = arr.reshape(2, 2, -1)  
  
print(newarr)

**Note:** We can not pass -1 to more than one dimension.

## 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)

**Note:** There are a lot of functions for changing the shapes of arrays in numpy flatten, ravel and also for rearranging the elements rot90, flip, fliplr, flipud etc. These fall under Intermediate to Advanced section of numpy.

# **NumPy Array Iterating**

## 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.

### **Example**

Iterate on the elements of the following 1-D array:

import numpy as np  
  
arr = np.array([1, 2, 3])  
  
for x in arr:  
  print(x)

## Iterating 2-D Arrays

In a 2-D array it will go through all the rows.

### **Example**

Iterate on the elements of the following 2-D array:

import numpy as np  
  
arr = np.array([[1, 2, 3], [4, 5, 6]])  
  
for x in arr:  
  print(x)

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

To return the actual values, the scalars, we have to iterate the arrays in each dimension.

### **Example**

Iterate on each scalar element of the 2-D array:

import numpy as np  
  
arr = np.array([[1, 2, 3], [4, 5, 6]])  
  
for x in arr:  
  for y in x:  
    print(y)

## Iterating 3-D Arrays

In a 3-D array it will go through all the 2-D arrays.

### **Example**

Iterate on the elements of the following 3-D array:

import numpy as np  
  
arr = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]])  
  
for x in arr:  
  print(x)

To return the actual values, the scalars, we have to iterate the arrays in each dimension.

### **Example**

Iterate down to the scalars:

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

## 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.

### **Example**

Iterate through the following 3-D array:

import numpy as np  
  
arr = np.array([[[1, 2], [3, 4]], [[5, 6], [7, 8]]])  
  
for x in np.nditer(arr):  
  print(x)

## 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'].

### **Example**

Iterate through the array as a string:

import numpy as np  
  
arr = np.array([1, 2, 3])  
  
for x in np.nditer(arr, flags=['buffered'], op\_dtypes=['S']):  
  print(x)

## Iterating With Different Step Size

We can use filtering and followed by iteration.

### **Example**

Iterate through every scalar element of the 2D array skipping 1 element:

import numpy as np  
  
arr = np.array([[1, 2, 3, 4], [5, 6, 7, 8]])  
  
for x in np.nditer(arr[:, ::2]):  
  print(x)

## Enumerated Iteration Using ndenumerate()

Enumeration means mentioning sequence number of somethings one by one.

Sometimes we require corresponding index of the element while iterating, the ndenumerate() method can be used for those usecases.

### **Example**

Enumerate on following 1D arrays elements:

import numpy as np  
  
arr = np.array([1, 2, 3])  
  
for idx, x in np.ndenumerate(arr):  
  print(idx, x)

### **Example**

Enumerate on following 2D array's elements:

import numpy as np  
  
arr = np.array([[1, 2, 3, 4], [5, 6, 7, 8]])  
  
for idx, x in np.ndenumerate(arr):  
  print(idx, x)

# **NumPy Joining Array**

## Joining NumPy Arrays

Joining means putting contents of two or more arrays in a single array.

In SQL we join tables based on a key, whereas in NumPy we join arrays by axes.

We pass a sequence of arrays that we want to join to the concatenate() function, along with the axis. If axis is not explicitly passed, it is taken as 0.

### **Example**

Join two arrays

import numpy as np  
  
arr1 = np.array([1, 2, 3])  
  
arr2 = np.array([4, 5, 6])  
  
arr = np.concatenate((arr1, arr2))  
  
print(arr)

### **Example**

Join two 2-D arrays along rows (axis=1):

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)

## Joining Arrays Using Stack Functions

Stacking is same as concatenation, the only difference is that stacking is done along a new axis.

We can concatenate two 1-D arrays along the second axis which would result in putting them one over the other, ie. stacking.

We pass a sequence of arrays that we want to join to the stack() method along with the axis. If axis is not explicitly passed it is taken as 0.

### **Example**

import numpy as np  
  
arr1 = np.array([1, 2, 3])  
  
arr2 = np.array([4, 5, 6])  
  
arr = np.stack((arr1, arr2), axis=1)  
  
print(arr)

## Stacking Along Rows

NumPy provides a helper function: hstack() to stack along rows.

### **Example**

import numpy as np  
  
arr1 = np.array([1, 2, 3])  
  
arr2 = np.array([4, 5, 6])  
  
arr = np.hstack((arr1, arr2))  
  
print(arr)

## Stacking Along Columns

NumPy provides a helper function: vstack()  to stack along columns.

### **Example**

import numpy as np  
  
arr1 = np.array([1, 2, 3])  
  
arr2 = np.array([4, 5, 6])  
  
arr = np.vstack((arr1, arr2))  
  
print(arr)

## Stacking Along Height (depth)

NumPy provides a helper function: dstack() to stack along height, which is the same as depth.

### **Example**

import numpy as np  
  
arr1 = np.array([1, 2, 3])  
  
arr2 = np.array([4, 5, 6])  
  
arr = np.dstack((arr1, arr2))  
  
print(arr)

# **NumPy Splitting Array**

## Splitting NumPy Arrays

Splitting is reverse operation of Joining.

Joining merges multiple arrays into one and Splitting breaks one array into multiple.

We use array\_split() for splitting arrays, we pass it the array we want to split and the number of splits.

### **Example**

Split the array in 3 parts:

import numpy as np  
  
arr = np.array([1, 2, 3, 4, 5, 6])  
  
newarr = np.array\_split(arr, 3)  
  
print(newarr)

**Note:** The return value is a list containing three arrays.

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

### **Example**

Split the array in 4 parts:

import numpy as np  
  
arr = np.array([1, 2, 3, 4, 5, 6])  
  
newarr = np.array\_split(arr, 4)  
  
print(newarr)

**Note:** We also have the method split() available but it will not adjust the elements when elements are less in source array for splitting like in example above, array\_split() worked properly but split() would fail.

## Split Into Arrays

The return value of the array\_split() method is an array containing each of the split as an array.

If you split an array into 3 arrays, you can access them from the result just like any array element:

### **Example**

Access the splitted arrays:

import numpy as np  
  
arr = np.array([1, 2, 3, 4, 5, 6])  
  
newarr = np.array\_split(arr, 3)  
  
print(newarr[0])  
print(newarr[1])  
print(newarr[2])

## Splitting 2-D Arrays

Use the same syntax when splitting 2-D arrays.

Use the array\_split() method, pass in the array you want to split and the number of splits you want to do.

### **Example**

Split the 2-D array into three 2-D arrays.

import numpy as np  
  
arr = np.array([[1, 2], [3, 4], [5, 6], [7, 8], [9, 10], [11, 12]])  
  
newarr = np.array\_split(arr, 3)  
  
print(newarr)

The example above returns three 2-D arrays.

Let's look at another example, this time each element in the 2-D arrays contains 3 elements.

### **Example**

Split the 2-D array into three 2-D arrays.

import numpy as np  
  
arr = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12], [13, 14, 15], [16, 17, 18]])  
  
newarr = np.array\_split(arr, 3)  
  
print(newarr)

The example above returns three 2-D arrays.

In addition, you can specify which axis you want to do the split around.

The example below also returns three 2-D arrays, but they are split along the row (axis=1).

### **Example**

Split the 2-D array into three 2-D arrays along rows.

import numpy as np  
  
arr = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12], [13, 14, 15], [16, 17, 18]])  
  
newarr = np.array\_split(arr, 3, axis=1)  
  
print(newarr)

An alternate solution is using hsplit() opposite of hstack()

### **Example**

Use the hsplit() method to split the 2-D array into three 2-D arrays along rows.

import numpy as np  
  
arr = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12], [13, 14, 15], [16, 17, 18]])  
  
newarr = np.hsplit(arr, 3)  
  
print(newarr)

**Note:** Similar alternates to vstack() and dstack() are available as vsplit() and dsplit().

# **NumPy Searching Arrays**

## Searching Arrays

You can search an array for a certain value, and return the indexes that get a match.

To search an array, use the where() method.

### **Example**

Find the indexes where the value is 4:

import numpy as np  
  
arr = np.array([1, 2, 3, 4, 5, 4, 4])  
  
x = np.where(arr == 4)  
  
print(x)

The example above will return a tuple: (array([3, 5, 6],)

Which means that the value 4 is present at index 3, 5, and 6.

### **Example**

Find the indexes where the values are even:

import numpy as np  
  
arr = np.array([1, 2, 3, 4, 5, 6, 7, 8])  
  
x = np.where(arr%2 == 0)  
  
print(x)

### **Example**

Find the indexes where the values are odd:

import numpy as np  
  
arr = np.array([1, 2, 3, 4, 5, 6, 7, 8])  
  
x = np.where(arr%2 == 1)  
  
print(x)

## Search Sorted

There is a method called searchsorted() which performs a binary search in the array, and returns the index where the specified value would be inserted to maintain the search order.

The searchsorted() method is assumed to be used on sorted arrays.

### **Example**

Find the indexes where the value 7 should be inserted:

import numpy as np  
  
arr = np.array([6, 7, 8, 9])  
  
x = np.searchsorted(arr, 7)  
  
print(x)

Example explained: The number 7 should be inserted on index 1 to remain the sort order.

The method starts the search from the left and returns the first index where the number 7 is no longer larger than the next value.

### **Search From the Right Side**

By default the left most index is returned, but we can give side='right' to return the right most index instead.

### **Example**

Find the indexes where the value 7 should be inserted, starting from the right:

import numpy as np  
  
arr = np.array([6, 7, 8, 9])  
  
x = np.searchsorted(arr, 7, side='right')  
  
print(x)

Example explained: The number 7 should be inserted on index 2 to remain the sort order.

The method starts the search from the right and returns the first index where the number 7 is no longer less than the next value.

### **Multiple Values**

To search for more than one value, use an array with the specified values.

### **Example**

Find the indexes where the values 2, 4, and 6 should be inserted:

import numpy as np  
  
arr = np.array([1, 3, 5, 7])  
  
x = np.searchsorted(arr, [2, 4, 6])  
  
print(x)

The return value is an array: [1 2 3] containing the three indexes where 2, 4, 6 would be inserted in the original array to maintain the order.

# **NumPy Sorting Arrays**

## Sorting Arrays

Sorting means putting elements in an ordered sequence.

Ordered sequence is any sequence that has an order corresponding to elements, like numeric or alphabetical, ascending or descending.

The NumPy ndarray object has a function called sort(), that will sort a specified array.

### **Example**

Sort the array:

import numpy as np  
  
arr = np.array([3, 2, 0, 1])  
  
print(np.sort(arr))

**Note:** This method returns a copy of the array, leaving the original array unchanged.

You can also sort arrays of strings, or any other data type:

### **Example**

Sort the array alphabetically:

import numpy as np  
  
arr = np.array(['banana', 'cherry', 'apple'])  
  
print(np.sort(arr))

### **Example**

Sort a boolean array:

import numpy as np  
  
arr = np.array([True, False, True])  
  
print(np.sort(arr))

## Sorting a 2-D Array

If you use the sort() method on a 2-D array, both arrays will be sorted:

### **Example**

Sort a 2-D array:

import numpy as np  
  
arr = np.array([[3, 2, 4], [5, 0, 1]])  
  
print(np.sort(arr))

# **NumPy Filter Array**

## Filtering Arrays

Getting some elements out of an existing array and creating a new array out of them is called filtering.

In NumPy, you filter an array using a boolean index list.

A boolean index list is a list of booleans corresponding to indexes in the array.

If the value at an index is True that element is contained in the filtered array, if the value at that index is False that element is excluded from the filtered array.

### **Example**

Create an array from the elements on index 0 and 2:

import numpy as np  
  
arr = np.array([41, 42, 43, 44])  
  
x = [True, False, True, False]  
  
newarr = arr[x]  
  
print(newarr)

The example above will return [41, 43], why?

Because the new array contains only the values where the filter array had the value True, in this case, index 0 and 2.

## Creating the Filter Array

In the example above we hard-coded the True and False values, but the common use is to create a filter array based on conditions.

### **Example**

Create a filter array that will return only values higher than 42:

import numpy as np  
  
arr = np.array([41, 42, 43, 44])  
  
# Create an empty list  
filter\_arr = []  
  
# go through each element in arr  
for element in arr:  
  # if the element is higher than 42, set the value to True, otherwise False:  
  if element > 42:  
    filter\_arr.append(True)  
  else:  
    filter\_arr.append(False)  
  
newarr = arr[filter\_arr]  
  
print(filter\_arr)  
print(newarr)

### **Example**

Create a filter array that will return only even elements from the original array:

import numpy as np  
  
arr = np.array([1, 2, 3, 4, 5, 6, 7])  
  
# Create an empty list  
filter\_arr = []  
  
# go through each element in arr  
for element in arr:  
  # if the element is completely divisble by 2, set the value to True, otherwise False  
  if element % 2 == 0:  
    filter\_arr.append(True)  
  else:  
    filter\_arr.append(False)  
  
newarr = arr[filter\_arr]  
  
print(filter\_arr)  
print(newarr)

## Creating Filter Directly From Array

The above example is quite a common task in NumPy and NumPy provides a nice way to tackle it.

We can directly substitute the array instead of the iterable variable in our condition and it will work just as we expect it to.

### **Example**

Create a filter array that will return only values higher than 42:

import numpy as np  
  
arr = np.array([41, 42, 43, 44])  
  
filter\_arr = arr > 42  
  
newarr = arr[filter\_arr]  
  
print(filter\_arr)  
print(newarr)

### **Example**

Create a filter array that will return only even elements from the original array:

import numpy as np  
  
arr = np.array([1, 2, 3, 4, 5, 6, 7])  
  
filter\_arr = arr % 2 == 0  
  
newarr = arr[filter\_arr]  
  
print(filter\_arr)  
print(newarr)