Numpy Introduction

NumPy is a Python library.NumPy is used for working with arrays. NumPy is short for "Numerical Python".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. NumPy arrays are stored at one continuous place in memory unlike lists, so processes can access and manipulate them very efficiently.

Import NumPy

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

In [2]:
```
import numpy as np
print(np.__version__)
```

```
1.19.2
```

NumPy Creating Arrays

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

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

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

Dimensions in Arrays

We can create a NumPy ndarray object by using the array() function. A dimension in arrays is one level of array depth (nested arrays). nested array: are arrays that have arrays as their elements.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

In [ ]:
```

```

In [ ]:
```

```

0-D Arrays

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

In [4]:
```
arr = np.array(42)
print(arr)
```

```
42
```

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.

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

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

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

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

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

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

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

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

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

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.

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

```
6
```

Negative Indexing

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

```
Last element from 2nd dim:  10
```

NumPy Array Slicing

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[1:5])
```

```
[2 3 4 5]
```

Use the step value to determine the step of the slicing

In [11]:
```
arr = np.array([1, 2, 3, 4, 5, 6, 7])
print(arr[1:5:2])
```

```
[2 4]
```

Slicing 2-D Arrays

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

In [12]:
```
arr = np.array([[1, 2, 3, 4, 5], [6, 7, 8, 9, 10]])
print(arr[0:2, 1:4])
```

```
[[2 3 4]
 [7 8 9]]
```

Data Types in NumPy

Below is a list of all data types in NumPy and the characters used to represent them.

1) i - integer 2) b - boolean 3) u - unsigned integer 4) f - float 5) c - complex float 6) m - timedelta 7) m- datetime 8) o - object 9) S-string 10) U- unicode string 11) V- fixed chunk of memory for other type (void)

In [13]:
```
arr = np.array([1, 2, 3, 4], dtype='S')
print(arr)
print(arr.dtype)
```

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

Converting Data Type on Existing Arrays

The best way to change the data type of an existing array, is to make a copy of the array with the astype() method. The astype() function creates a copy of the array, and allows you to specify the data type as a parameter. The data type can be specified using a string, like 'f' for float, 'i' for integer etc. or you can use the data type directly like float for float and int for integer.

In [14]:
```
arr = np.array([1.1, 2.1, 3.1])
newarr = arr.astype('i')
print(newarr)
print(newarr.dtype)
```

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

NumPy Array Copy vs 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 [15]:
```
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 [16]:
```
arr = np.array([1, 2, 3, 4, 5])
x = arr.view()
arr[0] = 42
print(arr)
print(x)
```

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

NumPy Array Shape

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

In [17]:
```
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)
```

NumPy Array Reshaping

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 [18]:
```
arr = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12])
newarr = arr.reshape(4, 3)
print(newarr)
```

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

Flattening the arrays

Flattening array means converting a multidimensional array into a 1D array. We can use reshape(-1) to do this.

In [19]:
```
arr = np.array([[1, 2, 3], [4, 5, 6]])
newarr = arr.reshape(-1)
print(newarr)
```

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

NumPy Array Iterating

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 [20]:
```
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)
```

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

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.

In [21]:
```
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
```

NumPy Joining Array

Joining means putting contents of two or more arrays in a single array. In NumPy we join arrays by axes.If axis is not explicitly passed, it is taken as 0. 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.

In [22]:
```
arr1 = np.array([1, 2, 3])
arr2 = np.array([4, 5, 6])
arr = np.stack((arr1, arr2), axis=1)
print(arr)
```

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

NumPy Splitting Array

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.

In [23]:
```
arr = np.array([[1, 2], [3, 4], [5, 6], [7, 8], [9, 10], [11, 12]])
newarr = np.array_split(arr, 3)
print(newarr)
```

```
[array([[1, 2],
       [3, 4]]), array([[5, 6],
       [7, 8]]), array([[ 9, 10],
       [11, 12]])]
```

NumPy 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.

In [24]:
```
arr = np.array([1, 2, 3, 4, 5, 6, 7, 8])
x = np.where(arr%2 == 1)
print(x)
```

```
(array([0, 2, 4, 6], dtype=int64),)
```

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.

In [25]:
```
arr = np.array([6, 7, 8, 9])
x = np.searchsorted(arr, 7)
print(x)
```

```
1
```

NumPy Sorting Arrays

Sorting means putting elements in an ordered sequence. The NumPy ndarray object has a function called sort(), that will sort a specified array.

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

```
[[2 3 4]
 [0 1 5]]
```

NumPy Filter Array

Getting some elements out of an existing array and creating a new array out of them is called filtering. 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.We can directly substitute the array instead of the iterable variable in our condition and it will work just as we expect it to.

In [4]:
```
import numpy as np
arr = np.array([41, 42, 43, 44])
filter_arr = arr > 42
newarr = arr[filter_arr]
print(filter_arr)
print(newarr)
```

```
[False False  True  True]
[43 44]
```

NumPy - Broadcasting

The term broadcasting refers to the ability of NumPy to treat arrays of different shapes during arithmetic operations. Arithmetic operations on arrays are usually done on corresponding elements. If two arrays are of exactly the same shape, then these operations are smoothly performed

In [5]:
```
a = np.array([1,2,3,4])
b = np.array([10,20,30,40])
c = a * b
c
```

Out[5]:
```
array([ 10,  40,  90, 160])
```

In [10]:
```
import numpy as np
a = np.array([[0.0,0.0,0.0],[10.0,10.0,10.0],[20.0,20.0,20.0],[30.0,30.0,30.0
b = np.array([1.0,2.0,3.0])
print ('First array:' )
print (a )
print ('Second array:' )
print (b )
print ('First Array + Second Array' )
print (a + b)
```

```
First array:
[[ 0.  0.  0.]
 [10. 10. 10.]
 [20. 20. 20.]
 [30. 30. 30.]]
Second array:
[1. 2. 3.]
First Array + Second Array
[[ 1.  2.  3.]
 [11. 12. 13.]
 [21. 22. 23.]
 [31. 32. 33.]]
```

In [ ]:
```
```

NumPy - Arithmetic Operations

In [9]:
```
a = np.arange(9, dtype = np.float_).reshape(3,3)
print ('First array:' )
print (a )
print ('Second array:' )
b = np.array([10,10,10])
print (b)
print ('Add the two arrays:' )
print (np.add(a,b) )
print ('Subtract the two arrays:' )
print (np.subtract(a,b) )
print ('Multiply the two arrays:' )
print (np.multiply(a,b) )
print ('Divide the two arrays:' )
print (np.divide(a,b))
```

```
First array:
[[0. 1. 2.]
 [3. 4. 5.]
 [6. 7. 8.]]
Second array:
[10 10 10]
Add the two arrays:
[[10. 11. 12.]
 [13. 14. 15.]
 [16. 17. 18.]]
Subtract the two arrays:
[[-10.  -9.  -8.]
 [ -7.  -6.  -5.]
 [ -4.  -3.  -2.]]
Multiply the two arrays:
[[ 0. 10. 20.]
 [30. 40. 50.]
 [60. 70. 80.]]
Divide the two arrays:
[[0.  0.1 0.2]
 [0.3 0.4 0.5]
 [0.6 0.7 0.8]]
```

In [ ]: