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.

Import Numpy

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

Create a NumPy ndarray Object

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

print(arr)

print(type(arr)) #Type of np.array
```

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

Dimensions in Arrays

0-d Arrays

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

print(arr)
```

```
42
```

1-D Array

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

print(arr)
```

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

2-D Array

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

print(arr)
```

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

3-D Array

In [ ]:
```
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]]]
```

Check Number of Dimensions?

In [ ]:
```
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)
```

```
0
1
2
3
```

Higher Dimensional Arrays

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

print(arr)
print('number of dimensions :', arr.ndim)
```

```
[[[[[1 2 3 4]]]]]
number of dimensions : 5
```

NumPy Array Indexing

Access 1-D Array Elements

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

print("arr[0]:",arr[0])
print("arr[1]:",arr[1])
print("arr[2] + arr[3]:",arr[2] + arr[3])
```

```
arr[0]: 1
arr[1]: 2
arr[2] + arr[3]: 7
```

Access 2-D Arrays

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

print('2nd element on 1st row: ', arr[0, 1]) #Access the element on the 1st row,
print('5th element on 2nd row: ', arr[1, 4]) #Access the element on the 2nd row,
```

```
2nd element on 1st row:  2
5th element on 2nd row:  10
```

Access 3-D Arrays

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

print("3rd element of the 2nd array of the 1st array:",arr[0, 1, 2]) #Access the
print("2nd element of the 2nd array of the 2nd array:",arr[1, 1, 1]) #Access the
```

```
3rd element of the 2nd array of the 1st array: 6
2nd element of the 2nd array of the 2nd array: 11
```

Negative Indexing

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

print('Last element from 2nd dim: ', arr[1, -1])
print('Lasts 2nd element from 1nd dim: ', arr[0, -2])
```

```
Last element from 2nd dim:  10
Lasts 2nd element from 1nd dim:  4
```

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

print(arr[1:5]) #Slice elements from index 1 to index 5 from the following array
print(arr[4:]) #Slice elements from index 4 to the end of the array.
print(arr[:4]) #Slice elements from the beginning to index 4 (not included).
```

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

Negative Slicing

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

```
print(arr[-3:-1]) #Slice from the index 3 from the end to index 1 from the end.
```

```
[5 6]
```

Step

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

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

print(arr[1:5:2]) # Return every other element from index 1 to index 5 with gap
print(arr[0:6:3]) # Return every other element from index 0 to index 6 with gap
print(arr[::2]) # Return every other element from the entire array with gap 2.
```

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

Slicing 2-D Arrays

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

print(arr[1, 1:4]) # From the second element, slice elements from index 1 to ind
print(arr[0:2, 2]) # From both elements, return index 2.
print(arr[0:2,1:3 ]) # From both elements, return index 1 to 2, this will return
```

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

Slicing 3-D Arrays

In [6]:
```
arr = np.array([[[10,34,34],[23,34,23]],
                [[45,34,56],[87,99,78]]])
print(arr[:,:,1:3]) # Slice the last column from each 3-D matrix.
```

```
[[[34 34]
  [34 23]]

 [[34 56]
  [99 78]]]
```

Data Types in Python

- Strings - used to represent text data, the text is given under quote marks. e.g. "ABCD"
- Integer - used to represent integer * numbers. e.g. -1, -2, -3
- float - used to represent real numbers. e.g. 1.2, 42.42
- Boolean - used to represent True or False.
- Complex - used to represent complex numbers. e.g. 1.0 + 2.0j, 1.5 + 2.5j

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)

Checking the Data Type of an Array

In [ ]:
```
arr = np.array([1, 2, 3, 4])
arr1 = np.array(['apple', 'banana', 'cherry', 'mango'])

print(arr.dtype)
print(arr1.dtype) # Get the data type of an array object.
```

```
int64
<U6
```

Creating Arrays With a Defined Data Type

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

print(arr)
print(arr.dtype)
arr1 = np.array([1, 2, 3, 4], dtype='i')

print(arr1)
print(arr1.dtype)
```

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

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 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 [ ]:
```
arr = np.array([1.1, 2.1, 3.1])

newarr = arr.astype('i')  #Convert into integer 32 bit
print(newarr)
print(newarr.dtype)

newarr = arr.astype(bool) # Convert into Boolean
print(newarr)
print(newarr.dtype)

newarr = arr.astype('S') # Convert into String 32bit
print(newarr)
print(newarr.dtype)
```

```
[1 2 3]
int32
[ True  True  True]
bool
[b'1.1' b'2.1' b'3.1']
|S32
```

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.

Copy

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

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

View

In [ ]:
```
arr = np.array([1, 2, 3, 4, 5])
x = arr.view()
arr[0] = 42
x[1] =78
print(arr)
print(x)
```

```
[42 78  3  4  5]
[42 78  3  4  5]
```

Check if Array Owns its 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 [ ]:
```
arr = np.array([1, 2, 3, 4, 5])

x = arr.copy()
y = arr.view()

print(x.base)
print(y.base)
```

```
None
[1 2 3 4 5]
```

Shape of an Array

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

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

print(arr.shape)
```

```
(2, 4)
```

In [ ]:
```
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.

Reshape From 1-D to 2-D

In [16]:
```
arr = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12])

newarr = arr.reshape(4, 3) # 4 x 3
newarr1 = arr.reshape(3, 4) # 3 X 4


print("4 X 3 matrix :-\n", newarr)
print()
print("3 X 4 matrix :-\n",newarr1)
```

```
4 X 3 matrix :-
 [[ 1  2  3]
 [ 4  5  6]
 [ 7  8  9]
 [10 11 12]]

3 X 4 matrix :-
 [[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]
```

Reshape From 1-D to 3-D

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

newarr = arr.reshape(2, 3, 2)

print(newarr)
```

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

 [[ 7  8]
  [ 9 10]
  [11 12]]]
```

Returns Copy or View?

In [ ]:
```
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]
```

Unknown Dimension

In [ ]:
```
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]]]
```

Flattening the arrays

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

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

newarr = arr.reshape(-1)

print(newarr)
```

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

NumPy Joining Array

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

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

arr = np.concatenate((arr1, arr2))

print(arr)
```

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

In [ ]:
```
arr1 = np.array([[1, 2], [3, 4]])

arr2 = np.array([[5, 6], [7, 8]])

arr = np.concatenate((arr1, arr2), axis=1) #Join two 2-D arrays along rows (axis
arr1 = np.concatenate((arr1, arr2), axis=0) #Join two 2-D arrays along rows (axi

print(arr)
print(arr1)
```

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

Stacking Along Rows

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

In [ ]:
```
arr1 = np.array([1, 2, 3])

arr2 = np.array([4, 5, 6])

arr = np.hstack((arr1, arr2))

print(arr)
```

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

Stacking Along Columns

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

In [ ]:
```
arr1 = np.array([1, 2, 3])

arr2 = np.array([4, 5, 6])

arr = np.vstack((arr1, arr2))

print(arr)
```

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

Stacking Along Height (depth)

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

In [ ]:
```
arr1 = np.array([1, 2, 3])

arr2 = np.array([4, 5, 6])

arr = np.dstack((arr1, arr2))

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])

newarr = np.array_split(arr, 2)
newarr1 = np.array_split(arr, 3)
newarr2 = np.array_split(arr, 4)
newarr3 = np.array_split(arr, 5)

print("Split into 2 parts:-",newarr)
print()
print("Split into 3 parts:-",newarr1)
print()
print("Split into 4 parts:-",newarr2)
print()
print("Split into 5 parts:-",newarr3)
print()
print("Access a 1st index:-",newarr[0])
print()
print("Access a 2nd index:-",newarr[1])
```

```
Split into 2 parts:- [array([1, 2, 3]), array([4, 5, 6])]

Split into 3 parts:- [array([1, 2]), array([3, 4]), array([5, 6])]

Split into 4 parts:- [array([1, 2]), array([3, 4]), array([5]), array([6])]

Split into 5 parts:- [array([1, 2]), array([3]), array([4]), array([5]), array
([6])]

Access a 1st index:- [1 2 3]

Access a 2nd index:- [4 5 6]
```

Splitting 2-D Arrays

In [25]:
```
arr = np.array([[1, 2], [3, 4], [5, 6], [7, 8], [9, 10], [11, 12]])

newarr = np.array_split(arr, 5)

print("Split into 5 parts:-",newarr)
```

```
Split into 5 parts:- [array([[1, 2],
       [3, 4]]), array([[5, 6]]), array([[7, 8]]), array([[ 9, 10]]), array([[11,
12]])]
```

Splitting 3-D Arrays

In [26]:
```
arr = np.array([[[1, 2], [3, 4]], [[5, 6], [7, 8]], [[9, 10], [11, 12]]])

newarr = np.array_split(arr, 5)

print("Split into 5 parts:-",newarr)
```

```
Split into 5 parts:- [array([[[1, 2],
        [3, 4]]]), array([[[5, 6],
        [7, 8]]]), array([[[ 9, 10],
        [11, 12]]]), array([], shape=(0, 2, 2), dtype=int64), array([], shape=(0,
2, 2), dtype=int64)]
```

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() met hod.

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

x = np.where(arr == 4) # Find the indexes where the value is 4:

print(x)
```

```
(array([3, 5, 6]),)
```

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

x = np.where(arr%2 == 0) #Find the indexes where the values are even:

print(x)
```

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

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 [ ]:
```
import numpy as np

arr = np.array([6, 7, 8, 9])

x = np.searchsorted(arr, 9)

print(x)
```

```
3
```

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.

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

x = np.searchsorted(arr, 7, side='right')

print(x)
```

```
2
```

Search From the Left Side

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

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

x = np.searchsorted(arr, 7, side='left')

print(x)
```

```
1
```

Multiple Values

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

x = np.searchsorted(arr, [2, 4, 6]) # Find the indexes where the values 2, 4, an

print(x)
```

```
[1 2 3]
```

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 [ ]:
```
arr = np.array([3, 2, 0, 1])

print(np.sort(arr))
```

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

Sorting a 2-D Array

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

print(np.sort(arr))
```

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

Sorting a 3-D array

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

```
[[[ 0  3  5]
  [ 3  8  9]]

 [[ 3  5  7]
  [ 1 10 11]]]
```

Sorting a String array

In [27]:
```
arr = np.array(['manngo','banana', 'cherry'])
print(np.sort(arr))
```

```
['banana' 'cherry' 'manngo']
```

Numpy Filter Array

- 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.

In [ ]:
```
arr = np.array([41, 42, 43, 44])

x = [True, False, True, False]

newarr = arr[x]

print(newarr) #eate an array from the elements on index 0 and 2:
```

```
[41 43]
```

Creating the filter array

In [ ]:
```
arr = np.array([41,42,43,44])

# Create an empty list
filter_arr = []

# Go Through each element in arr
for element in arr:
  if element > 42:
    filter_arr.append(True)
  else:
    filter_arr.append(False)

new_arr = arr[filter_arr]

print(filter_arr)
print(new_arr)
```

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

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

filter_arr = []

for element in arr:

  if element%2 == 0:
    filter_arr.append(True)
  else:
    filter_arr.append(False)

new_arr = arr[filter_arr]

print(filter_arr)
print(new_arr)
```

```
[False, True, False, True, False, True, False]
[2 4 6]
```

Creating Filter Directly From Array

In [ ]:
```
arr = np.array([41,42,43,44])

filter_arr = arr > 42

new_arr = arr[filter_arr]

print(filter_arr)
print(new_arr)
```

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