```
import numpy as np
```

> **Checking NumPy Version:**

The version string is stored under '**version**' attribute.

```
import numpy as np

print(np.__version__)
```

```
1.22.4
```

Create Arrays

> **1D Array:**

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

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

> **2D Array:**

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

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

> **Higher-Dimensional Arrays:**

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

print(arr)
```

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

> **Reshape:** Can create another array with different shape, form one array.

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

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

> **Flattening the arrays:**

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

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

print(newarr)
```

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

> **tolist():** Convert an array to list.

```
a = np.array([1,2,3,4,5,6,7,8,9])
l = a.tolist()
print(type(l), l)
```

```
<class 'list'> [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Attributes of ndarray

> **ndmin:** ndim attribute returns an integer that tells us how many dimensions the array have.

```
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
```

> **shape:** Return shape - (row, col) of the array.

```
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.shape)
print(b.shape)
print(c.shape)
print(d.shape)
```

```
()
(5,)
(2, 3)
(2, 2, 3)
```

> **size:** Return total number of elements in array = (row*col)

```
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.size)
print(b.size)
print(c.size)
print(d.size)
```

```
1
5
6
12
```

Access Array Elements

> **Iterating Arrays Using nditer():**

- Makes easy to iterate through each Scalar Element in the array, even for complex dimensions as well.

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

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

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

> **Iterating With Different Step Size:**

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

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

```
    1
    3
    5
    7
```

> **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.

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

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

```
    (0,) 1
    (1,) 2
    (2,) 3
```

1. Array Indexing

> **1-D Array Indexing:**

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

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

print(arr[0])
```

```
    1
```

> **2-D Array Indexing:**

- 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.

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

print('2nd element on 1st row: ', arr[0, 1])
```

```
    2nd element on 1st row:  2
```

3-D Array Indexing:

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

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

print(arr[0, 1, 2])
```

```
6
```

Negative Indexing:

- Use negative indexing to access an array from the end.

```
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
```

2. 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 start is not given, its considered 0.
- If end its is not given, considered length of array in that dimension.
- If step is not given, its considered 1.

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

print(arr[1:5])
print(arr[4:])
print(arr[:4])
```

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

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

Negative Slicing:

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

print(arr[-3:-1])
```

```
[5 6]
```

> **Slicing 2-D Arrays:** Slices or indices are separated by comma.

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

print(arr[1, 1:4])
print(arr[0:2, 2])
print(arr[0:2, 1:4])
```

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

Array Creation using NumPy Functions:

> **1. zeros:** Create array with all elements 0, with given dimensions.

```
a = np.zeros((5,4), dtype=int)
print(a)
```

```
[[0 0 0 0]
 [0 0 0 0]
 [0 0 0 0]
 [0 0 0 0]
 [0 0 0 0]]
```

> **2. ones:** Create array with all elements 1, with given dimensions.

```
a = np.ones((5,4), dtype=int)
print(a)
```

```
[[1 1 1 1]
 [1 1 1 1]
 [1 1 1 1]
 [1 1 1 1]
 [1 1 1 1]]
```

> **3. arange() Function:** Create a sequence of number using arange() function.

```
a = np.arange(10)
print(a)

b = np.arange(11,21,dtype='f')
print(b)
```

```
[0 1 2 3 4 5 6 7 8 9]
[11. 12. 13. 14. 15. 16. 17. 18. 19. 20.]
```

> **4. linspace:** Create 1-D array of linear space numbers / values, by default 50 linspace.

```
p = np.linspace(3,5)
print(p)

q = np.linspace(5,2)
print(q)

q = np.linspace(5,2,retstep=True)
print(q)
```

```
[3.         3.04081633 3.08163265 3.12244898 3.16326531 3.20408163
 3.24489796 3.28571429 3.32653061 3.36734694 3.40816327 3.44897959
 3.48979592 3.53061224 3.57142857 3.6122449  3.65306122 3.69387755
 3.73469388 3.7755102  3.81632653 3.85714286 3.89795918 3.93877551
 3.97959184 4.02040816 4.06122449 4.10204082 4.14285714 4.18367347
 4.2244898  4.26530612 4.30612245 4.34693878 4.3877551  4.42857143
 4.46938776 4.51020408 4.55102041 4.59183673 4.63265306 4.67346939
 4.71428571 4.75510204 4.79591837 4.83673469 4.87755102 4.91836735
 4.95918367 5.        ]
[5.         4.93877551 4.87755102 4.81632653 4.75510204 4.69387755
 4.63265306 4.57142857 4.51020408 4.44897959 4.3877551  4.32653061
 4.26530612 4.20408163 4.14285714 4.08163265 4.02040816 3.95918367
 3.89795918 3.83673469 3.7755102  3.71428571 3.65306122 3.59183673
 3.53061224 3.46938776 3.40816327 3.34693878 3.28571429 3.2244898
 3.16326531 3.10204082 3.04081633 2.97959184 2.91836735 2.85714286
 2.79591837 2.73469388 2.67346939 2.6122449  2.55102041 2.48979592
 2.42857143 2.36734694 2.30612245 2.24489796 2.18367347 2.12244898
 2.06122449 2.        ]
(array([5.        , 4.93877551, 4.87755102, 4.81632653, 4.75510204,
       4.69387755, 4.63265306, 4.57142857, 4.51020408, 4.44897959,
       4.3877551 , 4.32653061, 4.26530612, 4.20408163, 4.14285714,
       4.08163265, 4.02040816, 3.95918367, 3.89795918, 3.83673469,
       3.7755102 , 3.71428571, 3.65306122, 3.59183673, 3.53061224,
       3.46938776, 3.40816327, 3.34693878, 3.28571429, 3.2244898 ,
       3.16326531, 3.10204082, 3.04081633, 2.97959184, 2.91836735,
       2.85714286, 2.79591837, 2.73469388, 2.67346939, 2.6122449 ,
       2.55102041, 2.48979592, 2.42857143, 2.36734694, 2.30612245,
       2.24489796, 2.18367347, 2.12244898, 2.06122449, 2.        ]), -0.061224489795918366)
```

> **5. eye:** Return an 2-D array with 1's at diagonals and 0 elsewhere.

- **Syntax -** np.eye(R, C=None, k=0, dtype=?)
  - **R -** Number of Rows.
  - **C -** [Optional] Number of columns, by default, rows=cols.

```
e = np.eye(4)
print(e)
```

```
e = np.eye(4,3)
print(e)
```

```
[[1. 0. 0. 0.]
 [0. 1. 0. 0.]
 [0. 0. 1. 0.]
 [0. 0. 0. 1.]]
[[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]
 [0. 0. 0.]]
```

> **6. identity:** Same as eye(), but take only one argument. So, row=col.

```
d = np.identity(5, dtype='i')
print(d)
```

```
[[1 0 0 0 0]
 [0 1 0 0 0]
 [0 0 1 0 0]
 [0 0 0 1 0]
 [0 0 0 0 1]]
```

> **7. fromiter:** This routine is used to create a ndarray by using an iterable object. It returns a one-dimensional ndarray object.

```
list = [0,2,4,6]
it = iter(list)
x = np.fromiter(it, dtype = float)
print(x)
print(type(x))
```

```
[0. 2. 4. 6.]
<class 'numpy.ndarray'>
```

NumPy Function

> **insert():**
>
> - Insert values at given place.
> - changes are not permanent, need another values to store changes.

```
a = np.array([1,2,3,4,5])
a = np.insert(a, 2, 100)
print(a)
```

```
[  1   2 100   3   4   5]
```

> **append():**
>
> - Insert value at the end of the array

```
a = np.array([1,2,3,4,5])
a = np.append(a, 200)
print(a)
```

```
[  1   2   3   4   5 200]
```

> **ceil():**
>
> - Yield closest upper int value to the given float number.

```
a = np.array([1.2, 5.8, 9.4, 5.8, 12.5])
a = np.ceil(a)
print(a)
```

```
[ 2.  6. 10.  6. 13.]
```

> **floor():**
>
> - Yield closest lower int value to the given float number.

```
a = np.array([1.2, 5.8, 9.4, 5.8, 12.5])
a = np.floor(a)
print(a)
```

```
[ 1.  5.  9.  5. 12.]
```

> **around():**
>
> - Yield closest upper int value for values having decimal point>.5 and lower for point<0.5, for the given float number.

```
a = np.array([1.2, 5.8, 9.4, 5.8, 12.5])
a = np.around(a)
print(a)
```

```
[ 1.  6.  9.  6. 12.]
```

> **argmax():** Return index of the max element in the array,

```
a = np.array([1.2, 5.8, 9.4, 5.8, 12.5])
print(np.argmax(a))
```

```
4
```

> **argmin():** Return index of the min element in the array.

```
a = np.array([1.2, 5.8, 9.4, 5.8, 12.5])
print(np.argmin(a))

     0
```

> **where():**
>
> - Returns indices of values, for which the given condition is satisfied.
> - **Syntax -** np.where(condition, True(Replace with this value), False(Replace with this value))

```
a = np.array([1.2, 5.8, 9.4, 5.8, 12.5, 4.8])
print(np.where(a>5))

print(np.where(a>5, 10, 0))

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

> **size():** Return size of the array.

```
a = np.array([1.2, 5.8, 9.4, 5.8, 12.5, 4.8])
print(np.size(a))

     6
```

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.

> **copy():**

```
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]
```

> **view():**

```
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]
```

> **Check if Array Owns its Data:**

- Copies owns the data, and views does not own the data.
- 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.

- The copy returns **None**.
- The view returns the **original array**.

```
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]
```

NumPy Joining Array

> **Joining NumPy Arrays:**

- Joining means merging values of two or more arrays in a single array.
- 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.

> **Join row-wise:**

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

arr = np.concatenate((arr1, arr2), axis=0)
```

```
print(arr)
```

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

Join col-wise:

```
arr1 = np.array([[1, 2], [3, 4]])
arr2 = np.array([[5, 6], [7, 8]])

arr = np.concatenate((arr1, arr2), axis=1)

print(arr)
```

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

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.

Row-wise Stacking:

```
arr1 = np.array([1, 2, 3])
arr2 = np.array([4, 5, 6])
arr = np.stack((arr1, arr2), axis=0)
print(arr)
```

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

Column-wise Stacking:

```
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]]
```

Stacking Along Rows: NumPy provides a function: **hstack()** to stack along rows.

```
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 function: **vstack()** to stack along columns.

```
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 function: **dstack()** to stack along height, which is the same as depth.

```
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]]]
```

✓ 0s completed at 20:22