# INTRODUCTION TO NUMPY

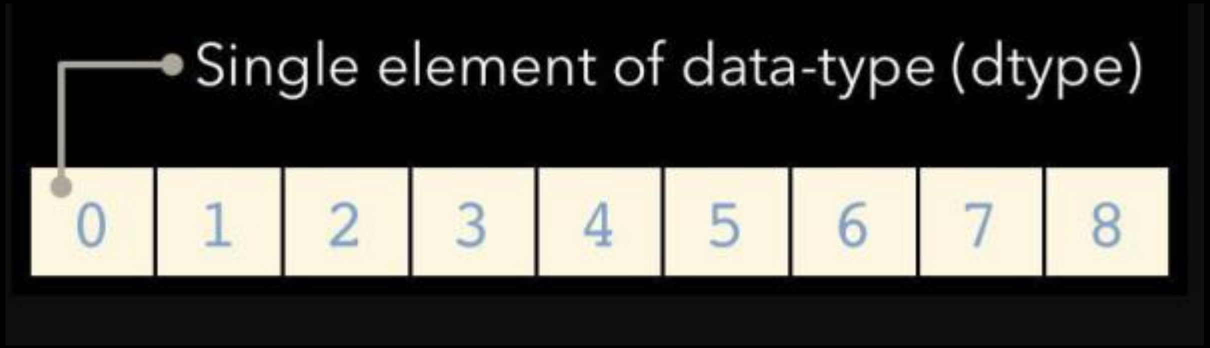
*By:*
*Sharmila Chidaravalli*
*Assistant Professor*
*Department of Information Science & Engineering*
*Global Academy of Technology*

## What is NumPy?

The NumPy library is the core library for scientific computing in Python.

It provides a high performance multidimensional array object and tools for working with these arrays.

The key to NumPy is the ndarray object, an *n*-dimensional array of homogeneous data types, with many operations being performed in compiled code for performance.



# What is NumPy?

There are several important differences between NumPy arrays and the standard Python sequences:

NumPy arrays have a fixed size. Modifying the size means creating a new array.

NumPy arrays must be of the same data type, but this can include Python objects.

More efficient mathematical operations than built-in sequence types.

**import numpy as np**

```
A=[[1,2,3],[4,5,6]]
print(A)
type(A)
```

**Output:**

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

```
A = np.array(A)
print(A)
type(A)
```

**Output:**

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

print(np . ndim(A))

Ans : ?

print(np . ndim(A)) Ans : 2

print(np. shape(A)) Ans : ?

```
print(np . ndim(A))
```

Ans : 2

```
print(np. shape(A))
```

Ans : (2,3)

```
rows = np.shape(A)[0]
columns = np.shape(A)[1]
print("number of rows = ",rows)
print("number of columns = ", columns)
```

**Output:**

```
number of rows = 2
number of columns = 3
```

```
import numpy as np
a = np.arange(15).reshape(3, 5)
print(a)
print(a.ndim)
print(a.shape)
print(a.dtype.name)
print(a.itemsize)
print(a.size)
b = np.array([6, 7, 8])
print(b)
type(b)
```

Ans:

```
[[ 0 1 2 3 4]
[ 5 6 7 8 9]
[10 11 12 13 14]]

2

 (3, 5)

int64

8 15

[6 7 8]

numpy.ndarray
```

## Array Creation

```
a = np.array([2,3,4])
print(a)
a.dtype
```

```
[2 3 4]
dtype('int64')
```

```
b = np.array([1.2, 3.5, 5.1])
print(b)
b.dtype
```

```
[1.2  3.5  5.1]
dtype('float64')
```

array transforms sequences of sequences into two-dimensional arrays, sequences of sequences of sequences into three-dimensional arrays, and so on.

```
b = np.array([(1.5,2,3), (4,5,6)])
print(b)
```

```
[[1.5  2.  3. ]
[4.     5.  6. ]]
```

The type of the array can also be explicitly specified at creation time:

```
c = np.array( [ [1,2], [3,4] ], dtype=complex )
print(c)
```

```
[[1.+0.j   2.+0.j]
3.+0.j      4.+0.j]]
```

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

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

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

The elements of an array are originally unknown, but its size is known.
Hence, NumPy offers several functions to create arrays with initial placeholder content. These minimize the necessity of growing arrays, an expensive operation.

The function **zeros** creates an array full of zeros, the function **ones** creates an array full of ones, and the function **empty** creates an array whose initial content is random and depends on the state of the memory.
By default, the dtype of the created array is float64.

```
print(np.zeros( (3,4) ))
```

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

```
print(np.ones( (2,3,4), dtype=np.int16 )  )
```

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

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

```
print(np.empty( (2,3) )   )
```

```
[[1.39069238e-309 1.39069238e-309 1.39069238e-309]
[1.39069238e-309 1.39069238e-309 1.39069238e-309]]
```

To create sequences of numbers, NumPy provides a function analogous to range that returns arrays instead of lists.

```
np.arange( 10, 30, 5 )                    [10, 15, 20, 25]

 np.arange( 0, 2, 0.3 )                   [ 0. , 0.3, 0.6, 0.9, 1.2, 1.5, 1.8]
```

When arange is used with floating point arguments, it is generally not possible to predict the number of elements obtained, due to the finite floating point precision. For this reason, it is usually better to use the function linspace that receives as an argument the number of elements that we want .

```
from numpy import pi
print(np.linspace( 0, 2, 9 ))
x = np.linspace( 0, 2*pi, 100 )
print(x)
f = np.sin(x)
print(f)
```

Ans: ?

## *Basic Operations*

Arithmetic operators on arrays apply *elementwise*.
A new array is created and filled with the result.

```
a = np.array( [20,30,40,50] )
print(a)

b = np.arange( 4 )
print(b)

c = a-b
print(c)

print(b**2)

print(10*np.sin(a))

print(a<35)
```

Ans: ?

Unlike in many matrix languages,
 the product operator * operates element wise in NumPy arrays.
The matrix product can be performed using the @ operator (in python >=3.5) or the dot function or method

```
 A= np.array( [[1,1],
            [0,1]] )
print(A)

B = np.array( [[2,0],
          [3,4]] )
print(B)

print("The Element wise product")
print(A * B)

print("The Matrix Product")
print(A @ B)

print("The Matrix Product using dot function")
print(A.dot(B))
```

Ans: ?

```
a=np. array([1,2,3])
b=np.array([(1.5,2,3),(4,5,6)],dtype=float)
c=np. array([[(1.5,2,3),(4,5,6)],[(3,2,1),(4,5,6)]], dtype=float)
print("The 1D",a)
print("The 2D",b)
print("The 3D",c)
d=np.arange(10,25,5)
print(d)
e=np. full((2,2),7)
print("The full array")
print(e)
f=np. eye(3)
print(“The 3 *3 identity matrix")
print(f)
print("the random array")
print(np.random.random((2,2)))
```

Ans: ?

```
print("The subtraction of a& b :")
print(np.subtract(a,b))
```

*Similarly try*

```
np.add(b,a)

np.divide(a,b)

np.multiply(a,b)

np.exp(b)

np.sqrt(b)

np.sin(a)

np.cos(b)

np.log(a)
```

Also try comparison operations

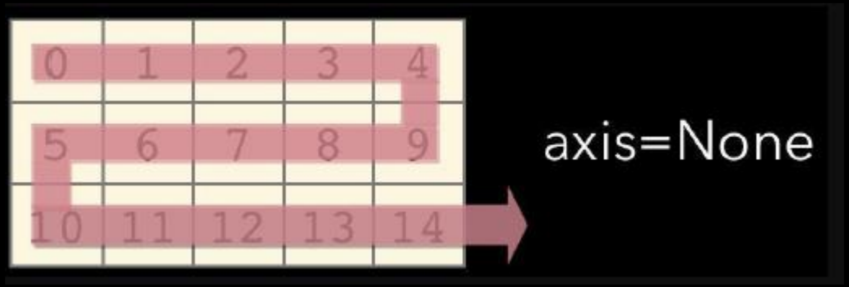
```
a == b

a < 2

np.array_equal(a,b)
```

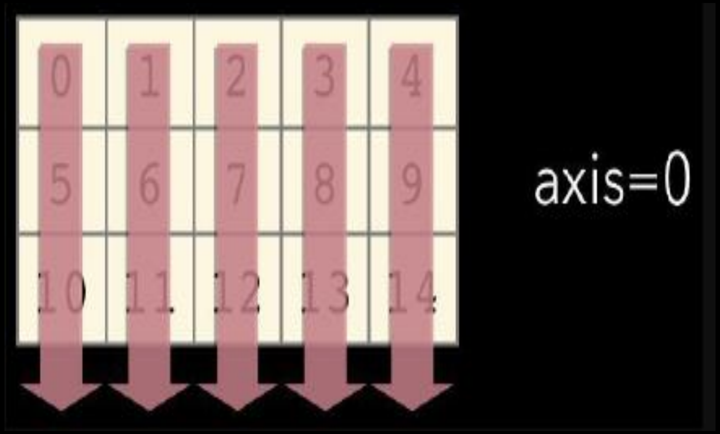
## *Aggregate Functions*

```
print(b.sum())
print(np.sum(b))
```

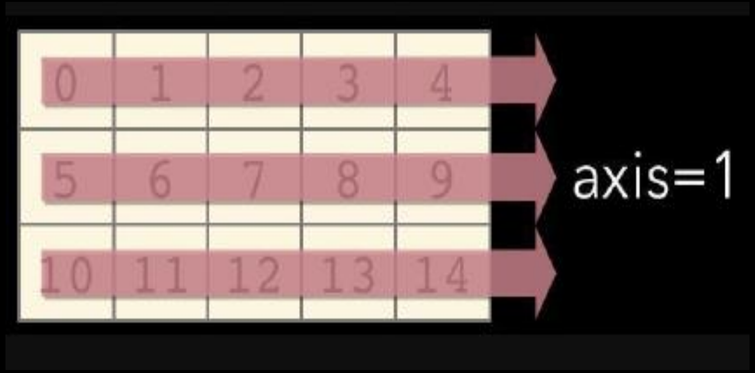


Similarly try

```
a.sum()
a.min()
b.max(axis = 0)
b.cumsum(axis = 1)
a.mean()
b.median()
a.corrcoef()
np.std(b)
```



Ans: ?



## *Copying Arrays*

```
h=a.view()
print(h)

C=np.copy(b)
print(C)

h=a.copy()
print(h)
```

## *Sorting Arrays*

```
b=np.array([5,7,2,4,1,9,6,0])
print(b)
print(np.sort(b))

a.sort()

c.sort(axis=0)
```

## *Subsetting*

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

| | | |
|---|---|---|
| 1 | 2 | 3 |

b=np.array([(1.5,2,3),(4,5,6)],dtype=float)

| | | |
|---|---|---|
| 1.5 | 2 | 3 |
| 4 | 5 | 6 |

a[2]

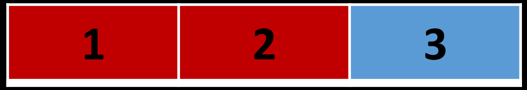
| | | |
|---|---|---|
| 1 | 2 | 3 |

b[1,2]

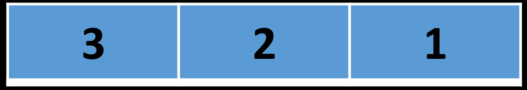
| | | |
|---|---|---|
| 1.5 | 2 | 3 |
| 4 | 5 | 6 |

## *Slicing*

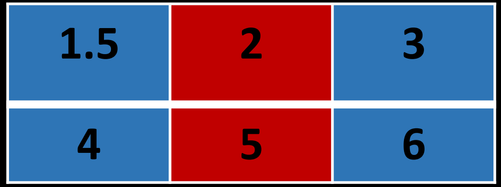
a[ 0 : 2 ]



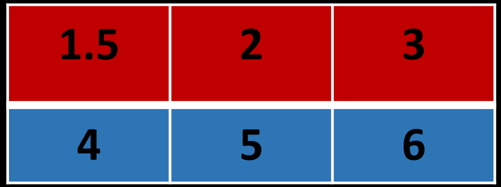
a[ : : -1 ]



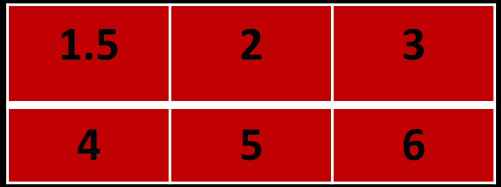
b[ 0 : 2 ,1 ]

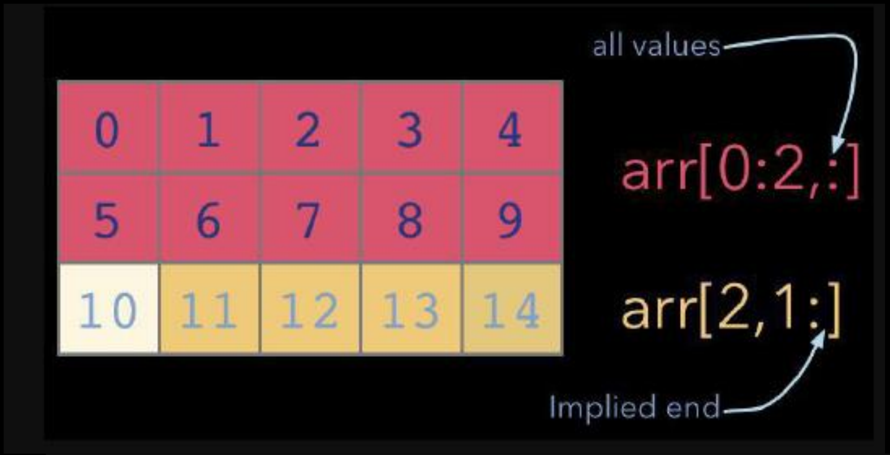


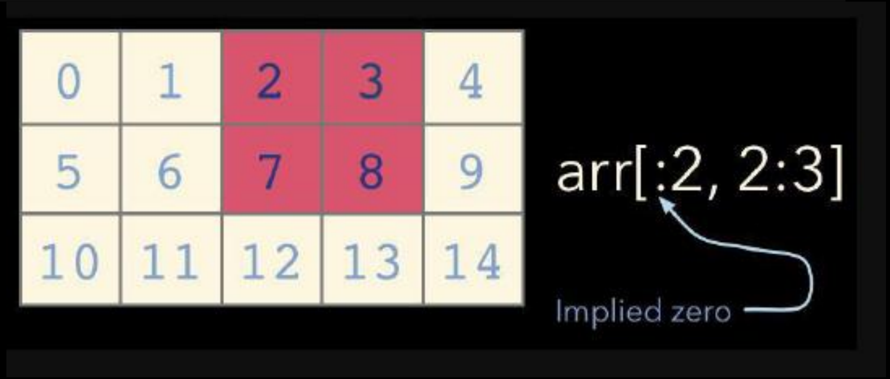
b[ : 1 ]

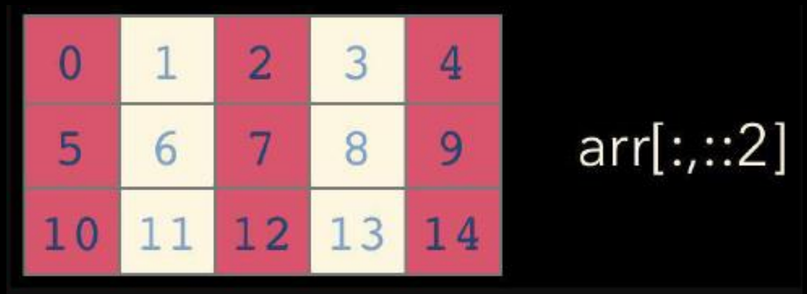


b[ : 2 ]

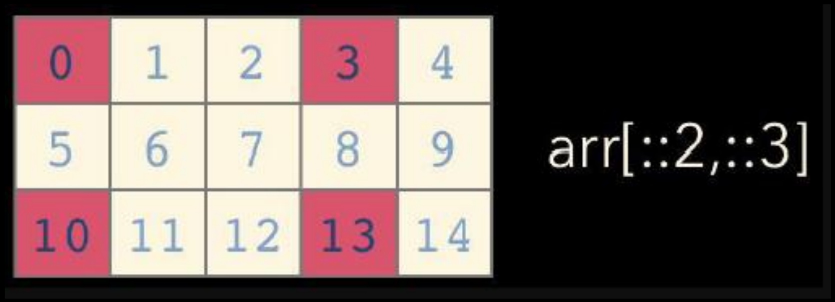



all values
arr[0:2,:]
arr[2,1:]
Implied end


arr[:2, 2:3]
Implied zero

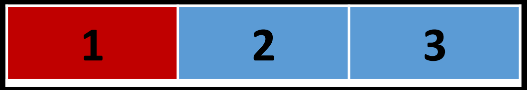

arr[:,::2]

Ans: ?


arr[::2,::3]

## *Indexing*

a[a<2]



b[[1,0,1,0],[0,1,2,0]]

```
[ 4.  2.  6.  1.5 ]
```

b[[1, 0, 1, 0]] [:,[0,1,2,0]]

```
[[ 4.  5.  6.  4. ]
[1.5   2.  3. 1.5]
[4.     5.  6.  4. ]
[1.5   2.  3.  1.5]]
```

## *Transposing*

```
i = np.transpose(b)
print(i)
i.T
```

## *Changing Array Shape*

`b.ravel()` Flatten the array

`g.reshape(3,-2)` Reshape, but don't change data

```
[[-0.5 0. 0. ]
[-3. -3. -3. ]]
```

```
array([[-0.5, 0. ],
       [ 0. , -3. ],
       [-3. , -3. ]])
```

After reshape

```
A = [[1,2,3],[4,5,6]]
print(A)
type(A)

A = np.asarray(A)
print(A)
type(A)
```

```
print("Maximum element is ",np.max(A))
print(np.min(A))
print(np.mean(A))
print(np.median(A))
print(np.std(A))
B = np.transpose(A)
print(B)
C = np.reshape(A,(1,6))
print(C)
rows = np.shape(C)[0]
columns = np.shape(C)[1]
print("number of rows = ",rows)
print("number of columns = ", columns)
print(A[0])
print(A[:,0])
print(A[:,1])
```

```
print(A)
print(A[:,0])
print(A[:,1])
print(A[:,2])
print(np.sum(A[:,0]))
print(np.sum(A[0,:]))
```