Numpy

Create an array and Type of Data

```
import numpy as np
```

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

```
array([[1, 2, 3],
       [4, 5, 6]])
```

Create an array using the array() function

```
d = np.array(((1,2,3),(4,5,6)))
d
```

```
array([[1, 2, 3],
       [4, 5, 6]])
```

the array() function can accept tuples and sequences of tuples

```
g = np.array([['a','b'],['c','d']])
g
```

```
array([['a', 'b'],
       ['c', 'd']], dtype='<U1')
```

Numpy arrays are designed to contain a variety of data types, not just integers

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

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

The array() function also accepts tuples and interconnected list

```
f = np.array([[1,2,3],[4,5,6]], dtype=complex)
f
```

```
array([[1.+0.j, 2.+0.j, 3.+0.j],
       [4.+0.j, 5.+0.j, 6.+0.j]])
```

Using the dtype option as argument of the array() function

Dtype Option and Intrinsic Creation of an Array

- We can use the `dtype` option as argument to explicitly define the dtype object.
- The `zero()` function creates a full array of zeros with dimensions defined by the shape argument.
- The `ones()` function creates an array full of ones.
- The `arange()` function generates NumPy arrays with numerical sequences.
- The `reshape()` function divides a linear array in different parts in the manner specified by the shape argument.
- The `linespace()` function takes as its two arguments the initial and the end values of the sequence, but the third argument defines the number of elements into which we want the interval to split.
- Another way to create arrays is using the `random()` function

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

```
array([[0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.]])
```

```
np.ones((3,3))
```

```
array([[1., 1., 1.],
       [1., 1., 1.],
       [1., 1., 1.]])
```

```
np.arange(0,10)
```

```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

```
np.arange(0,12,3)
```

```
array([0, 3, 6, 9])
```

```
np.arange(0,12).reshape(3,4)
```

```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

```
np.linspace(0,10,5)
```

```
array([ 0. ,  2.5,  5. ,  7.5, 10. ])
```

```
np.random.random(3)
```

```
array([0.34321252, 0.70708786, 0.78976411])
```

```
np.random.random((3,3))
```

```
array([[0.64299655, 0.33192893, 0.80073073],
       [0.74630879, 0.08996707, 0.34759507],
       [0.40177263, 0.61981972, 0.27107058]])
```

Arithmetic

Adding and multiplying an array by a scalar

```
a = np.arange(4)
a
```

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

```
a + 4
```

```
array([4, 5, 6, 7])
```

```
a * 2
```

```
array([0, 2, 4, 6])
```

`Element-wise` operation:

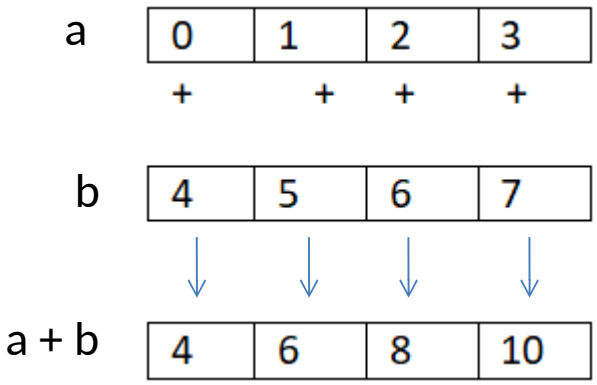
operators are applied only between corresponding elements

```
b = np.arange(4,8)
b
```

```
array([4, 5, 6, 7])
```

```
a + b
```

```
array([ 4,  6,  8, 10])
```




```
a-b
```

```
array([-4, -4, -4, -4])
```

```
a * b
```

Arithmetic operators for Functions

We can multiply the array by the sine or square root of the elements of array `b`

```
a * np.sin(b)
```

```
array([-0.        , -0.95892427, -0.558831  ,  1.9709598 ])
```

```
a * np.sqrt(b)
```

```
array([0.        , 2.23606798, 4.89897949, 7.93725393])
```

```
A = np.arange(0,9).reshape(3,3)
A
```

```
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])
```

```
B = np.ones((3,3))
B
```

```
array([[1., 1., 1.],
       [1., 1., 1.],
       [1., 1., 1.]])
```

`Element-wise` multidimensional operation

```
A * B
```

```
array([[0., 1., 2.],
       [3., 4., 5.],
       [6., 7., 8.]])
```

The Matrix Product

- Many tools for data analysis use the `*` operator as a matrix product when it is applied to two matrices.
- Using NumPy, this kind of product is indicated by the `dot()` function.
- This operation is not element-wise

```
A = np.arange(0,9).reshape(3,3)
A
```

```
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])
```

```
B = np.ones((3,3))
B
```

```
array([[1., 1., 1.],
       [1., 1., 1.],
       [1., 1., 1.]])
```

```
A * B
```

```
array([[0., 1., 2.],
       [3., 4., 5.],
       [6., 7., 8.]])
```

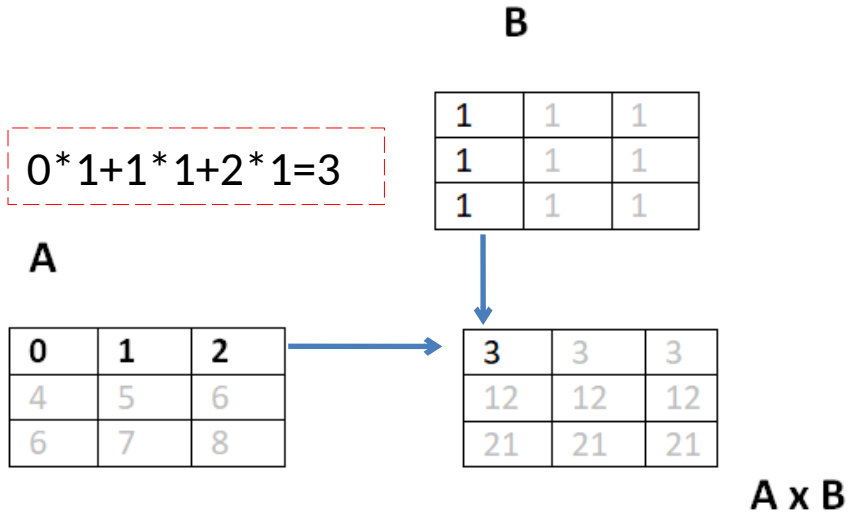
```
np.dot(A,B)
```

```
array([[ 3.,  3.,  3.],
       [12., 12., 12.],
       [21., 21., 21.]])
```

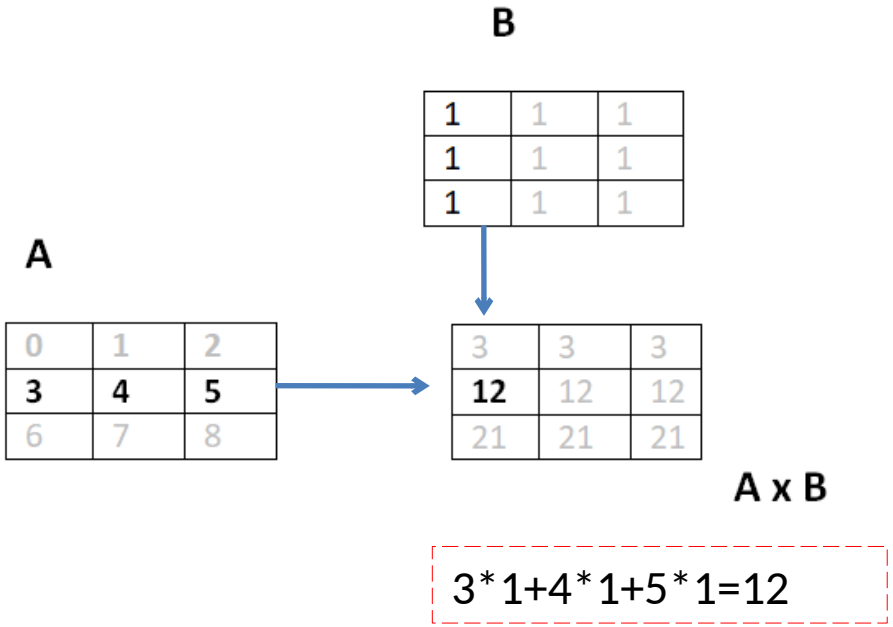
The result at each position is the sum of the products of each element of the corresponding row of the first matrix with the corresponding element of the column of the second matrix.

The Matrix Product

- Using NumPy, this kind of product is indicated by the `dot()` function.



- This operation is not element-wise



Transpose - Trace - Inverse

We can use the **transpose()** function to flip the original matrix.

This is achieve by switching its rows with its columns. Transpose does not modify the original array. This example shows two ways of doing it.

```
A = np.array([[15,35,45],[60,59,67],[50,78,99]])

#transpose
A.T
print(A)

#transpose
A.transpose()
print(A)
```

The ***trace*** of a square *matrix* is the sum of the diagonal elements. Use **trace()** function

```
import numpy as np

A = np.array([[15,35,45],[60,59,67],[50,78,99]])

# trace
print(np.trace(A))
```

The **inverse** of a matrix A is a matrix that, when multiplied by A results in the identity. The notation for this inverse matrix is A^{-1}. This code finds the invers of A.

```
import numpy as np

A = np.array([[15,35,45],[60,59,67],[50,78,99]])

#Inverse
print(np.linalg.inv(A))
```

Increment and Decrement Operators

- In Python, there are no operators called ++ or - - to increase or decrease.
- In Python, to increase we use the +=
- To decrease, we use the −=
- These are useful if we want to change the values in an array without generating a new array.

```
a = np.arange(4)
a
```

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

```
a += 1
a
```

```
array([2, 3, 4, 5])
```

```
a -= 1
a
```

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

```
a += 4
a
```

```
array([4, 5, 6, 7])
```

```
a *= 2
a
```

```
array([ 8, 10, 12, 14])
```

Shape manipulation

- It is possible to convert a one-dimensional array into a matrix using the `reshape()` function.
- The `reshape()` returns a new array and therefore create new objects.
- We want to modify the object by modifying the shape, we have to assign a tuple containing the new dimensions directly to its shape attributes.
- The `ravel()` function is used to convert a two-dimensional array into a one-dimensional array
- The `transpose()` function is used to invert the columns with the rows

```
a = np.random.random(12)
a
```

```
array([0.93648146, 0.49712723, 0.23628688, 0.57393036, 0.52174171,
       0.94516367, 0.59237128, 0.96787483, 0.20880308, 0.29318431,
       0.32277472, 0.9270486 ])
```

```
A = a.reshape(3,4)
A
```

```
array([[0.93648146, 0.49712723, 0.23628688, 0.57393036],
       [0.52174171, 0.94516367, 0.59237128, 0.96787483],
       [0.20880308, 0.29318431, 0.32277472, 0.9270486 ]])
```

```
a.shape = (3,4)
a
```

```
array([[0.93648146, 0.49712723, 0.23628688, 0.57393036],
       [0.52174171, 0.94516367, 0.59237128, 0.96787483],
       [0.20880308, 0.29318431, 0.32277472, 0.9270486 ]])
```

```
a = a.ravel()
a
```

```
array([0.93648146, 0.49712723, 0.23628688, 0.57393036, 0.52174171,
       0.94516367, 0.59237128, 0.96787483, 0.20880308, 0.29318431,
       0.32277472, 0.9270486 ])
```

```
a.shape = (12)
a
```

```
array([0.93648146, 0.49712723, 0.23628688, 0.57393036, 0.52174171,
       0.94516367, 0.59237128, 0.96787483, 0.20880308, 0.29318431,
       0.32277472, 0.9270486 ])
```

```
A.transpose()
```

```
array([[0.93648146, 0.52174171, 0.20880308],
       [0.49712723, 0.94516367, 0.29318431],
       [0.23628688, 0.59237128, 0.32277472],
       [0.57393036, 0.96787483, 0.9270486 ]])
```

- You can merge multiple arrays to form a new one that contains all of the arrays. This concept is known as `stacking` in Numpy.
- The `vstack()` function (vertical stacking) combines the second array as new rows of the first array
- For the `hstack()` function (horizontal stacking), the second array is added to the column of the first array.
- The `column_stack()` function and `row_stack()` function can also be used for stacking however, these are generally used for one-dimensional arrays which are stacked as column or rows in order to form a new two dimensional array

```
A = np.ones((3,3))
B = np.zeros((3,3))
np.vstack((A,B))
```

```
array([[1., 1., 1.],
       [1., 1., 1.],
       [1., 1., 1.],
       [0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.]])
```

```
np.hstack((A,B))
```

```
array([[1., 1., 1., 0., 0., 0.],
       [1., 1., 1., 0., 0., 0.],
       [1., 1., 1., 0., 0., 0.]])
```

```
a = np.array([0,1,2])
b = np.array([3,4,5])
c = np.array([6,7,8])
np.column_stack((a,b,c))
```

```
array([[0, 3, 6],
       [1, 4, 7],
       [2, 5, 8]])
```

```
np.row_stack((a,b,c))
```

```
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])
```

Array Manipulation – Splitting Arrays

- In Numpy, we have the `hsplit()` function to divide the array horizontally.
- The `vsplit()` function to split it vertically

```
A = np.arange(16).reshape((4,4))
A
```

```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15]])
```

4x4 matrix A

```
[B,C] = np.hsplit(A,2)
B
```

```
array([[ 0,  1],
       [ 4,  5],
       [ 8,  9],
       [12, 13]])
```

```
C
```

```
array([[ 2,  3],
       [ 6,  7],
       [10, 11],
       [14, 15]])
```

Matrix A is split horizontally into two 2x4 (2 columns by 4 rows) matrices, B and C. 2x4 means 2 columns by 4 rows.

```
[B,C] = np.vsplit(A,2)
B
```

```
array([[0, 1, 2, 3],
       [4, 5, 6, 7]])
```

```
C
```

```
array([[ 8,  9, 10, 11],
       [12, 13, 14, 15]])
```

Matrix A is split vertically into two 4x2 matrices, B and C. 4x2 means (4 columns by 2 rows)

Array Manipulation – Splitting Arrays

- The `split()` function allows us to split the array into nonsymmetrical parts.
- Passing the array as an argument, you have to also specify the indexes of the part to be divided.
- The option `axis =1` means the indexes will be the columns and the option `axis = 0` means they will be the row indexes.

- E.g. If we want to divide the matrix into three parts, the first part will include the first column, the second part will include the second and third column and the third part

```
A = np.arange(16).reshape((4,4))
A
```

```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15]])
```

```
[A1,A2,A3] = np.split(A,[1,3],axis=1)
A1
```

```
array([[ 0],
       [ 4],
       [ 8],
       [12]])
```

```
A2
```

```
array([[ 1,  2],
       [ 5,  6],
       [ 9, 10],
       [13, 14]])
```

```
A3
```

```
array([[ 3],
       [ 7],
       [11],
       [15]])
```

```
[A1,A2,A3] = np.split(A,[1,3],axis=0)
A1
```

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

```
A2
```

```
array([[ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

```
A3
```

```
array([[12, 13, 14, 15]])
```

Array Indexing

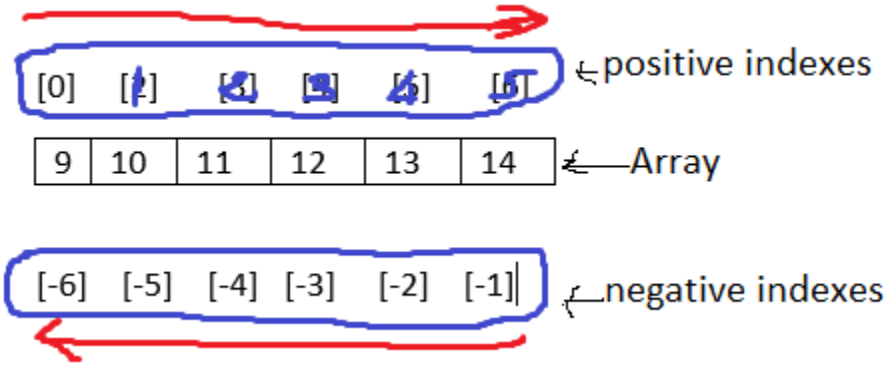
Uses square brackets ([ ]) to index the element of the array.

Index allows you to extract a value, select items or assign a new value.

```
import numpy as np
a = np.arange(9, 15)
print(a)
print(a[4])
```

```
[ 9 10 11 12 13 14]
13
```

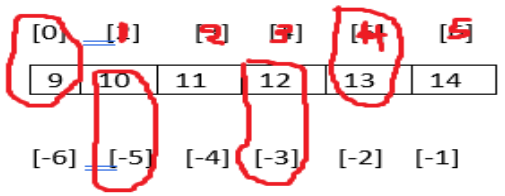
You can also use negative index with numpy. Negative indexes cause the final element to move gradually toward the first element. In this case the first element is the one with the more negative value.



To select many items at the same time, pass the array of indexes in square brackets

```
print(a[[0, -3, 4, -5]])
```

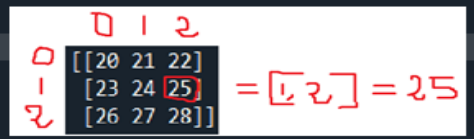
```
[ 9 12 13 10]
```



In this bidimensional matrix, we extract the element of the third column in the second row by inserting the pair [1, 2].

```
A = np.arange(20,29).reshape((3,3))
print(A)
print(A[1,2])
```

```
[[20 21 22]
 [23 24 25]
 [26 27 28]]
```



Array slicing

In numpy, we can create new arrays by slicing an existing array.

We must use the slice syntax which is a sequence of numbers separated by colons (:) within square brackets.

```
import numpy as np
a = np.arange(9, 15)
print(a)                    [ 9 10 11 12 13 14]

print(a[2:5])               [11 12 13]

print(a[1:5:2])             [10 12]

print(a[::2])               [ 9 11 13]

print(a[:5:2])              [ 9 11 13]

print(a[:5:])             [ 9 10 11 12 13]
```

Extract from 3rd element to 5th

Extract an item, skip a specific number, extract next & skip

Extract first item then every second item until max index of array

Extract first 5 items

In two-dimensional array, the slicing syntax is defined separately for the rows and columns

```
A = np.arange(20,29).reshape((3,3))
print(A)                    [[20 21 22]
                             [23 24 25]
                             [26 27 28]]

print(A[0,:])               [20 21 22]

print(A[0:2, 0:2])          [[20 21]
                             [23 24]]

print(A[[0,2], 0:2])        [[20 21]
                             [26 27]]

print(A[:, 0])            [20 23 26]
```

Extract first row only

Extract smaller matrix with contiguous rows or columns

Extract smaller matrix with non-contiguous rows or columns

Extract all values in first column

Iterating an Array

- For a two-dimensional matrix iteration is like applying two nested loops with the `for` construct. The first loop scan the rows of arrays and the second loop scans the columns. A `for loop` will always scan according to the first axis

```
import numpy as np
A = np.arange(20,29).reshape((3,3))

print()
for row in A:

    print(row)
```

```
[20 21 22]

[23 24 25]

[26 27 28]
```

Here we are using the for loop on **`A.flat`** to make an iteration element by element.

```
....
for item in A.flat:
    print(item)
```

```
20
21
22
23
24
25
26
27
28
```

Better, we can leave it to numpy to manage the iteration using the **`apply_along_axis()`** function which takes 3 arguments: the aggregate function, the axis on which the iteration will be applied, and the array. Example calculate the average values first by column then by row.

```
print(np.apply_along_axis(np.mean, axis = 0, arr=A))

print(np.apply_along_axis(np.mean, axis = 1, arr=A))
```

```
[23. 24. 25.]
[21. 24. 27.]
```

We can use a predefined function in numpy or define our own function. The **`ufunc`** which performs one iteration element by element.

```
def foo(x):
    return x/4
print(np.apply_along_axis(foo, axis = 0, arr=A))


print(np.apply_along_axis(foo, axis = 1, arr=A))
```

```
[[5.   5.25 5.5 ]
 [5.75 6.   6.25]
 [6.5  6.75 7.  ]]


[[5.   5.25 5.5 ]
 [5.75 6.   6.25]
 [6.5  6.75 7.  ]]
```

```python
import numpy as np
print(np.__version__)
```

Will get the numpy version

```python
import numpy as np
x = np.eye(3)
print(x)
```

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

Creates a 3x3 matrix – diagonal element are 1 and the rest 0

```python
import numpy as np
x = np.array([10, 2, 30, 45])
print("Original array:")
print(x)
print("Test if NONE of the elements of the array is zero:")
print(np.all(x))
```

Test whether none of the elements of a given array is zero

```python
import numpy as np
x = np.array([10, 2, 30, 45])
print("Original array:")
print(x)
print("Test if any of the elements of the array is non-zero:")
print(np.any(x))
```

Test whether any of the elements of the array is non-zero

convert a given array into a list and then convert it into a list again

```python
import numpy as np
myarr = [[10, 25], [40, 44]]
x = np.array(myarr)
myarr2 = x.tolist()
print(myarr == myarr2)
```

```python
import numpy as np
x = np.array([45, 67, 23])
y = np.array([56, 23, 89])
print("Original numbers:")
print(x)
print(y)
print("Comparison - greater")
print(np.greater(x, y))
print("Comparison - greater_equal")
print(np.greater_equal(x, y))
print("Comparison - less")
print(np.less(x, y))
print("Comparison - less_equal")
print(np.less_equal(x, y))
```

create an element-wise comparison (greater, greater_equal, less and less_equal) of two given arrays

```python
import numpy as np
import matplotlib.pyplot as plt
# Compute the x and y coordinates for points on a sine curve
x = np.arange(0, 5 * np.pi, 0.2)
y = np.sin(x)
print("Plot the points using matplotlib:")
plt.plot(x, y)
plt.show()
```

This program computes the x and y coordinates for points on a sine curve and plot the points using matplotlib

Reading and Writing Array Data on files

```
import numpy as np
import os
x = np.arange(9).reshape(3, 3)
print("Original array:")
print(x)
header = 'col1 col2 col3 col4'
np.savetxt('temp.txt', x, fmt="%d", header=header)
print("After loading, content of the text file:")
result = np.loadtxt('temp.txt')
print(result)
```

Using numpy savetxt() and load txt() functions to save a given array to a text file and load it

```
Original array:
[[0 1 2]
 [3 4 5]
 [6 7 8]]
After loading, content of the text file:
[[0. 1. 2.]
 [3. 4. 5.]
 [6. 7. 8.]]
```

Numpy save() and load() functions allows us to save and retrieve data stored in binary format.

```
"creates a 4x4 matrix of values between 0 and 1"

data = ([[0.01282092, 0.74680588, 0.81164726, 0.09672095],
 [0.61533233, 0.81034377, 0.24010536, 0.26836702],
 [0.61709381, 0.7612087,  0.39313723, 0.2594927 ],
 [0.73323143, 0.65083341, 0.2683259,  0.30541111]])

"Save the matrix values in a file called save_test_data"
np.save('save_test_data', data)

"retrieved the saved values. Notice the extension at the end of the file"
load_test_data = np.load('save_test_data.npy')
"display the saved "
print(load_test_data)
```

```
[[0.01282092 0.74680588 0.81164726 0.09672095]
 [0.61533233 0.81034377 0.24010536 0.26836702]
 [0.61709381 0.7612087  0.39313723 0.2594927 ]
 [0.73323143 0.65083341 0.2683259  0.30541111]]
```

Copies or Views of Objects

Assigning array1 to array2 is just another way to call array1.

When we slice an array, the object returned is a **view** of the original array. We are pointing to the same object. If the first array change the second array also change.

Use the **copy()** function to generate a complete and distinct array With this function, any changes made in the original array won't affect the second array.

```
import numpy as np
array1 = np.array([1,2,3,4])
array2 = array1
print(array2)

array1[2 ] = 0
print(array2)



array3 = array1[0:2]
print(array3)
array1[0] = 0
print(array3)

array4 = np.array([11,22,33,44])
array5 = array4.copy()
print(array5)
array4[0] =0
print(array5)
```