***NumPy (Numerical Python) is an open source Python library that's used in almost every field of science and engineering. It's the universal standard for working with numerical data in Python, and it's at the core of the scientific Python and PyData ecosystems.The NumPy API is used extensively in Pandas, SciPy, Matplotlib, scikit-learn, scikit-image and most other data science and scientific Python packages.The NumPy library contains multidimensional array and matrix data structures***

## What is NumPy?

NumPy stands for numeric python which is a python package for the computation and processing of the multidimensional and single dimensional array elements.

**Travis Oliphant** created NumPy package in 2005 by injecting the features of the ancestor module Numeric into another module Numarray.

## Why use NumPy?

NumPy arrays are faster and more compact than Python lists. An array consumes less memory and is convenient to use. NumPy uses much less memory to store data and it provides a mechanism of specifying the data types. This allows the code to be optimized even further.

To work with ndarrays, we need to load the numpy library. It is standard practice to load numpy with the alias "np" like so:

In [1]:

```
import numpy as np
```

The "as np" after the import statement lets us access the numpy library's functions using the shorthand "np."

Create an ndarray by passing a list to np.array() function:

In [2]:

```
my_list = [1, 2, 3, 4]              # Define a list

my_array = np.array(my_list)        # Pass the list to np.array()

type(my_array)                      # Check the object's type
```

Out[2]:

```
numpy.ndarray
```

In [2]:

```python
'''
shape : gives row by columns
ndim  : rank of the array

The number of dimensions is the rank of the array; the shape of an array is a
tuple of integers giving the size of the array along each dimension.
'''

array1D = np.array( [1,2,3,4] )
print('1D array \n', array1D) # 1D array | vector
print('Shape     : ', array1D.shape) # (4,)
print('Rank      : ', array1D.ndim)  # 1
print('Size      : ', array1D.size)  # 4
print('DatA Type : ', array1D.dtype) # int


print('--------------------------------------------')

array2D = np.array( [ [1.,2.,3.,4.], [4.,3.,2.,1.] ] )
print('2D array \n',array2D) # 2D array | matrix
print('Shape     : ', array2D.shape) # (2,4)
print('Rank      : ', array2D.ndim)  # 2
print('Size      : ', array2D.size)  # 8
print('DatA Type : ', array2D.dtype) # float


print('--------------------------------------------')

array3D = np.array( [ [[1,2,3,4]], [[-1,-2,-3,-4]], [[1,2,3,4]] ] )
print('3D array \n', array3D)
print('Shape     : ', array3D.shape) # (3, 1, 4)
print('Rank      : ', array3D.ndim)  # 3
print('Size      : ', array3D.size)  # 12
print('DatA Type : ', array3D.dtype) # int
```

```
1D array
 [1 2 3 4]
Shape     :  (4,)
Rank      :  1
Size      :  4
DatA Type :  int32
```

```
--------------------------------------------
2D array
 [[1. 2. 3. 4.]
 [4. 3. 2. 1.]]
Shape     :  (2, 4)
Rank      :  2
Size      :  8
DatA Type :  float64
--------------------------------------------
3D array
 [[[ 1  2  3  4]]

 [[-1 -2 -3 -4]]

 [[ 1  2  3  4]]]
Shape     :  (3, 1, 4)
Rank      :  3
Size      :  12
DatA Type :  int32
```

In [3]:

```
'''
arange  : Return evenly spaced values within a given interval.     (doc words)
reshape : Gives a new shape to an array without changing its data. (doc words)
'''

numbers = np.arange(10) # It will create a 1D numpy array
print(numbers)          # 0,1,2,3,4,5,6,7,8,9
print(numbers.dtype)    # int
print(type(numbers))    # numpy.ndarray

print('--------------------------------------------')

reshape_number = numbers.reshape(2,5) # It'll create a 2D numpy array.
print(reshape_number)         # [[0 1 2 3 4] [5 6 7 8 9]]
print(reshape_number.dtype)   # int
print(type(reshape_number))   # numpy.ndarray

print('--------------------------------------------')

array2D = np.arange(20).reshape(4,5) # Create 2D array with shape (4,5) from 0 to 19
print('2D array \n',array2D)         # 2D array | matrix
print('Shape     : ', array2D.shape) # (4,5)
print('Rank      : ', array2D.ndim)  # 2
print('Size      : ', array2D.size)  # 20
print('DatA Type : ', array2D.dtype) # int

print('--------------------------------------------')

array3D = np.arange(20).reshape(2, 2, 5) # Create 3D array with shape (2,2,5) from 0 to 19
print('3D array \n',array3D)         # 2D array | matrix
print('Shape     : ', array3D.shape) # (2, 2, 5) | (channel , width, height) ; we've two 2 by 5 matrix
print('Rank      : ', array3D.ndim)  # 3
print('Size      : ', array3D.size)  # 20
print('DatA Type : ', array3D.dtype) # int
```

```
[0 1 2 3 4 5 6 7 8 9]
int32
<class 'numpy.ndarray'>
--------------------------------------------
[[0 1 2 3 4]
```

```
 [5 6 7 8 9]]
int32
<class 'numpy.ndarray'>
---------------------------------------------
2D array
 [[ 0  1  2  3  4]
 [ 5  6  7  8  9]
 [10 11 12 13 14]
 [15 16 17 18 19]]
Shape     :  (4, 5)
Rank      :  2
Size      :  20
DatA Type :  int32
---------------------------------------------
3D array
 [[[ 0  1  2  3  4]
  [ 5  6  7  8  9]]

 [[10 11 12 13 14]
  [15 16 17 18 19]]]
Shape     :  (2, 2, 5)
Rank      :  3
Size      :  20
DatA Type :  int32
```

In [4]:

```
all_zeros = np.zeros((2,2))   # Create an array of all zeros
print('All Zeros \n' ,all_zeros)     # Prints "[[ 0.  0.]
                                     #          [ 0.  0.]]"

print('------------------------------------------------')

all_ones = np.ones((1,2))       # Create an array of all ones
print('All Ones \n', all_ones)  # Prints "[[ 1.  1.]]"

print('------------------------------------------------')

filled_array = np.full((2,2), 3)                          # Create a constant array
print('Filled with specified valued \n', filled_array)    # Prints "[[ 3.  3.]
                                                          #          [ 3.  3.]]"
print('------------------------------------------------')

identity_mat = np.eye(2)                       # Create a 2x2 identity matrix
print('Identity Matrix \n', identity_mat)  # Prints "[[ 1.  0.]
                                           #          [ 0.  1.]]"

print('------------------------------------------------')

random_normal_distro = np.random.random((2,2))  # Create an array filled with random values
print('Normal Distribution \n', random_normal_distro)

print('------------------------------------------------')

evenly_spaced_ranged_number = np.linspace(1,3,10) # range 1 to 3, generate 10 digit with evely spaced
print('Evely spaced number in givend range \n', evenly_spaced_ranged_number)

print('------------------------------------------------')

linspace_reshape = np.linspace(1,3,10).reshape(2,5)
print('2D array \n',linspace_reshape)         # 2D array | matrix
print('Shape              : ', linspace_reshape.shape) # (2, 5)
print('Rank               : ', linspace_reshape.ndim)  # 2
print('Size               : ', linspace_reshape.size)  # 10
print('DatA Type          : ', linspace_reshape.dtype) # float
print('Converted Data Type : ', linspace_reshape.astype('int64').dtype) # convert float to int
print('2D array \n',linspace_reshape.astype('int64')) # But this will truncated numbers after decimal

```

```
All Zeros
 [[0. 0.]
 [0. 0.]]
--------------------------------------------
All Ones
 [[1. 1.]]
--------------------------------------------
Filled with specified valued
 [[3 3]
 [3 3]]
--------------------------------------------
Identity Matrix
 [[1. 0.]
 [0. 1.]]
--------------------------------------------
Normal Distribution
 [[0.5989092  0.85605031]
 [0.34966726 0.40335178]]
--------------------------------------------
Evely spaced number in givend range
 [1.         1.22222222 1.44444444 1.66666667 1.88888889 2.11111111
 2.33333333 2.55555556 2.77777778 3.        ]
--------------------------------------------
2D array
 [[1.         1.22222222 1.44444444 1.66666667 1.88888889]
 [2.11111111 2.33333333 2.55555556 2.77777778 3.        ]]
Shape               :  (2, 5)
Rank                :  2
Size                :  10
DatA Type           :  float64
Converted Data Type :  int64
2D array
 [[1 1 1 1 1]
 [2 2 2 2 3]]
```

# What is an array?

An array is a central data structure of the NumPy library. An array is a grid of values and it contains information about the raw data, how to locate an element, and how to interpret an element. It has a grid of elements that can be indexed in various ways.

An array can be indexed by a tuple of nonnegative integers, by booleans, by another array, or by integers. The rank of the array is the number of dimensions. The shape of the array is a tuple of integers giving the size of the array along each dimension.

**OR**

1. Arrays in NumPy: NumPy’s main object is the homogeneous multidimensional array.
2. It is a table of elements (usually numbers), all of the same type, indexed by a tuple of positive integers.
3. In NumPy dimensions are called axes. The number of axes is rank.
4. NumPy’s array class is called ndarray. It is also known by the alias array.

One way we can initialize NumPy arrays is from Python lists, using nested lists for two- or higher-dimensional data.

Numpy offers several ways to index into arrays. We may want to select a subset of our data or individual elements. Most common ways are:

- Slicing
- Integer Array Indexing / Fnacy Indexing
- Boolean Indexing

***Slicing Like in Python lists, NumPy arrays can be sliced.***

In [5]:
```
array2D = np.arange(0,40,2).reshape(4,5)
print(array2D) # shape : (4,5)
```

```
[[ 0  2  4  6  8]
 [10 12 14 16 18]
 [20 22 24 26 28]
 [30 32 34 36 38]]
```

In [6]:
```
'''
Use slicing to pull out the subarray from the orignial array.
Le's say we want to get following sub-array from array2D.
This located at row (1,2) and column (1,2).

[12 14]
[22 24]

So, we need to do somthing like array2D[row-range, column-range]. Note that,
while indexing we need to range 1 step more, as we do
in np.arange(0,10) <- go 0 to 9 but not 10.

'''
# and columns 1 and 2; b is the following array of shape (2, 2):

sliced_array_1 = array2D[1:3, 1:3] # look we set 1:3 <- go 1 to 2 but not include 3
print(sliced_array_1)

print('--------------------------------------------')

sliced_array_2 = array2D[:, 1:4] # The 'bare' slice [:] will asign to all values in an array
print(sliced_array_2)

print('--------------------------------------------')

sliced_array_3 = array2D[:4, 2:] # row: 0 to 3 ; column: 2 to all
print(sliced_array_3)

'''
More practice. array2D:

[[ 0  2  4  6  8]
 [10 12 14 16 18]
 [20 22 24 26 28]
 [30 32 34 36 38]]

Let's get some specific portion.

1. [16 18],
   [26 28]

```

```
2. [20 22 24],
   [30 32 34]

3. [14 16],
   [24 26],
   [34 36]
'''

print('--------------------------------------------')

sliced_array_4 = array2D[1:3, 3:] # row: 1 to 2 ; column: 3 to all
print('1 \n', sliced_array_4)

print('--------------------------------------------')

sliced_array_5 = array2D[2:, 0:3] # row: 2 to all ; column: 0 to 2
print('2 \n', sliced_array_5)

print('--------------------------------------------')

sliced_array_6 = array2D[1:, 2:4] # row: 1 to all ; column: 2 to 3
print('3 \n', sliced_array_6)
```

```
[[12 14]
 [22 24]]
--------------------------------------------
[[ 2  4  6]
 [12 14 16]
 [22 24 26]
 [32 34 36]]
--------------------------------------------
[[ 4  6  8]
 [14 16 18]
 [24 26 28]
 [34 36 38]]
--------------------------------------------
1
 [[16 18]
 [26 28]]
--------------------------------------------
2
```

```
 [[20 22 24]
 [30 32 34]]
--------------------------------------------
3
 [[14 16]
 [24 26]
 [34 36]]
```

In [ ]:

```

```

```
Example:
[[ 1, 2, 3],
 [ 4, 2, 5]]
Here,
rank = 2 (as it is 2-dimensional or it has 2 axes)
first dimension(axis) length = 2, second dimension has length = 3
overall shape can be expressed as: (2, 3)
```

In [7]:

```
# Python program to demonstrate
# basic array characteristics

# Creating array object
arr = np.array( [[ 1, 2, 3],
                 [ 4, 2, 5]] )

# Printing type of arr object
print("Array is of type: ", type(arr))

# Printing array dimensions (axes)
print("No. of dimensions: ", arr.ndim)

# Printing shape of array
print("Shape of array: ", arr.shape)

# Printing size (total number of elements) of array
print("Size of array: ", arr.size)

# Printing type of elements in array
print("Array stores elements of type: ", arr.dtype)

```

```
Array is of type:  <class 'numpy.ndarray'>
No. of dimensions:  2
Shape of array:  (2, 3)
Size of array:  6
Array stores elements of type:  int32
```

In [8]:

```
import numpy as np

a = np.array([1,2,3,4])
        #OR
b = np.array([[1,2,3,4],[5,6,7,7],[8,9,10,11]])
print(b[0])
```

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

We can access the elements in the array using square brackets. When you're accessing elements, remember that indexing in NumPy starts at 0. That means that if you want to access the first element in your array, you'll be accessing element "0".

**"ndarray," which is shorthand for "N-dimensional array." An N-dimensional array is simply an array with any number of dimensions. You might also hear 1-D, or one-dimensional array, 2-D, or two-dimensional array, and so on. The NumPy ndarray class is used to represent both matrices and vectors. A vector is an array with a single dimension (there's no difference between row and column vectors), while a matrix refers to an array with two dimensions. For 3-D or higher dimensional arrays, the term tensor is also commonly used.**

**How To Make An "Empty" NumPy Array?**

In [9]:
```
# Create an array of ones
np.ones((3,4))

# Create an array of zeros
np.zeros((2,3,4),dtype=np.int16)

# Create an array with random values
np.random.random((2,2))

# Create an empty array
np.empty((3,2))

# Create a full array
np.full((2,2),7)

# Create an array of evenly-spaced values
np.arange(10,25,5)

# Create an array of evenly-spaced values
np.linspace(0,2,9)
```

Out[9]:
```
array([0.  , 0.25, 0.5 , 0.75, 1.  , 1.25, 1.5 , 1.75, 2.  ])
```

## Array creation: There are various ways to create arrays in NumPy.

- For example, you can create an array from a regular Python list or tuple using the array function. The type of the resulting array is deduced from the type of the elements in the sequences.

- Often, 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.
- For example: np.zeros, np.ones, np.full, np.empty, etc.
- To create sequences of numbers, NumPy provides a function analogous to range that returns arrays instead of lists.
- **arange**: returns evenly spaced values within a given interval. step size is specified.
- **linspace**: returns evenly spaced values within a given interval. num no. of elements are returned.
- **Reshaping array**: We can use reshape method to reshape an array. Consider an array with shape (a1, a2, a3, …, aN). We can reshape and convert it into another array with shape (b1, b2, b3, …, bM). The only required condition is:
- a1 x a2 x a3 … x aN = b1 x b2 x b3 … x bM . (i.e original size of array remains unchanged.)
- **Flatten array**: We can use flatten method to get a copy of array collapsed into one dimension. It accepts order argument. Default value is ‘C’ (for row-major order). Use ‘F’ for column major order.

Note: Type of array can be explicitly defined while creating array.

In [10]:

```python
# Python program to demonstrate
# array creation techniques
import numpy as np

# Creating array from list with type float
a = np.array([[1, 2, 4], [5, 8, 7]], dtype = 'float')
print ("Array created using passed list:\n", a)

# Creating array from tuple
b = np.array((1 , 3, 2))
print ("\nArray created using passed tuple:\n", b)

# Creating a 3X4 array with all zeros
c = np.zeros((3, 4))
print ("\nAn array initialized with all zeros:\n", c)

# Create a constant value array of complex type
d = np.full((3, 3), 6, dtype = 'complex')
print ("\nAn array initialized with all 6s."
            "Array type is complex:\n", d)

# Create an array with random values
e = np.random.random((2, 2))
print ("\nA random array:\n", e)

# Create a sequence of integers
# from 0 to 30 with steps of 5
f = np.arange(0, 30, 5)
print ("\nA sequential array with steps of 5:\n", f)

# Create a sequence of 10 values in range 0 to 5
g = np.linspace(0, 5, 10)
print ("\nA sequential array with 10 values between"
       "0 and 5:\n", g)

# Reshaping 3X4 array to 2X2X3 array
arr = np.array([[1, 2, 3, 4],
                [5, 2, 4, 2],
                [1, 2, 0, 1]])

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

```python

print ("\nOriginal array:\n", arr)
print ("Reshaped array:\n", newarr)

# Flatten array
arr = np.array([[1, 2, 3], [4, 5, 6]])
flarr = arr.flatten()

print ("\nOriginal array:\n", arr)
print ("Fattened array:\n", flarr)

```

```
Array created using passed list:
 [[1. 2. 4.]
 [5. 8. 7.]]

Array created using passed tuple:
 [1 3 2]

An array initialized with all zeros:
 [[0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]]

An array initialized with all 6s.Array type is complex:
 [[6.+0.j 6.+0.j 6.+0.j]
 [6.+0.j 6.+0.j 6.+0.j]
 [6.+0.j 6.+0.j 6.+0.j]]

A random array:
 [[0.86453441 0.42679317]
 [0.42547355 0.64569687]]

A sequential array with steps of 5:
 [ 0  5 10 15 20 25]

A sequential array with 10 values between0 and 5:
 [0.         0.55555556 1.11111111 1.66666667 2.22222222 2.77777778
 3.33333333 3.88888889 4.44444444 5.        ]

Original array:
```

```
 [[1 2 3 4]
 [5 2 4 2]
 [1 2 0 1]]
Reshaped array:
 [[[1 2 3]
  [4 5 2]]

 [[4 2 1]
  [2 0 1]]]

Original array:
 [[1 2 3]
 [4 5 6]]
Fattened array:
 [1 2 3 4 5 6]
```

#### Creating a Numpy Array

Arrays in Numpy can be created by multiple ways, with various number of Ranks, defining the size of the Array. Arrays can also be created with the use of various data types such as lists, tuples, etc. The type of the resultant array is deduced from the type of the elements in the sequences. Note: Type of array can be explicitly defined while creating the array.

In [11]:
```
# python program for creation of array
import numpy as np

# creating Rank 1 array
arr = np.array([1,2,3])
print("array with rank 1 : \n",arr)

# Creating a rank 2 Array
arr = np.array([[1,2,3],
               [4,5,6]])
print("arrray with rank 2 : \n",arr)

## Creating an array from tuple
arr = np.array((1, 3, 2))
print("\nArray created using "
      "passed tuple:\n", arr)
```

```
array with rank 1 :
 [1 2 3]
arrray with rank 2 :
 [[1 2 3]
 [4 5 6]]

Array created using passed tuple:
 [1 3 2]
```

**Array Indexing**: Knowing the basics of array indexing is important for analysing and manipulating the array object. NumPy offers many ways to do array indexing.

- Slicing: Just like lists in python, NumPy arrays can be sliced. As arrays can be multidimensional, you need to specify a slice for each dimension of the array.
- Integer array indexing: In this method, lists are passed for indexing for each dimension. One to one mapping of corresponding elements is done to construct a new arbitrary array.
- Boolean array indexing: This method is used when we want to pick elements from array which satisfy some condition.

## Array Indexing

Elements in NumPy arrays can be accessed by indexing. Indexing is an operation that pulls out a select set of values from an array. The index of a value in an array is that value's location within the array. There is a difference between the value and where the value is stored in an array.

An array with 3 values is created in the code section below.

In [12]:
```
a = np.array([2,4,6])
print(a)
```

```
[2 4 6]
```

The array above contains three values: 2, 4 and 6. Each of these values has a different index.

Remember counting in Python starts at 0 and ends at n-1.

The value 2 has an index of 0. We could also say 2 is in location 0 of the array. The value 4 has an index of 1 and the value 6 has an index of 2. The table below shows the index (or location) of each value in the array.

| Index (or location) | Value |
|---|---|
| 0 | 2 |
| 1 | 4 |
| 2 | 6 |

Individual values stored in an array can be accessed with indexing.

The general form to index a NumPy array is below:

```
<value> = <array>[index]`
```

Where is the value stored in the array, is the array object name and [index] specifies the index or location of that value.

In the array above, the value 6 is stored at index 2.

In [13]:
```
a = np.array([2,4,6])
print(a)
value = a[2]
print(value)
```

```
[2 4 6]
6
```

## Multi-dimensional Array Indexing

Multi-dimensional arrays can be indexed as well. A simple 2-D array is defined by a list of lists.

In [14]:
```
a = np.array([[2,3,4],[6,7,8]])
print(a)
```

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

```
Values in a 2-D array can be accessed using the general notation below:

<value> = <array>[row,col]
```

Where is the value pulled out of the 2-D array and [row,col] specifies the row and column index of the value. Remember Python counting starts at 0, so the first row is row zero and the first column is column zero.

We can access the value 8 in the array above by calling the row and column index [1,2]. This corresponds to the 2nd row (remember row 0 is the first row) and the 3rd column (column 0 is the first column).

In [15]:

```
a = np.array([[2,3,4],[6,7,8]])
print(a)
value = a[1,2]
print(value)
```

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

## Assigning Values with Indexing

Array indexing is used to access values in an array. And array indexing can also be used for assigning values of an array.

The general form used to assign a value to a particular index or location in an array is below:

```
<array>[index] = <value>
```

Where is the new value going into the array and [index] is the location the new value will occupy.

The code below puts the value 10 into the second index or location of the array a.

In [16]:

```
a = np.array([2,4,6])
a[2] = 10
print(a)
```

```
[ 2  4 10]
```

```
Values can also be assigned to a particular location in a 2-D arrays using the form:

<array>[row,col] = <value>
```

The code example below shows the value 20 assigned to the 2nd row (index 1) and 3rd column (index 2) of the array.

In [17]:

```python
a = np.array([[2,3,4],[6,7,8]])
print(a)
print("***************")

a[1,2] = 20
print(a)

```

```
[[2 3 4]
 [6 7 8]]
***************
[[ 2  3  4]
 [ 6  7 20]]
```

## Array Slicing

Multiple values stored within an array can be accessed simultaneously with array slicing. To pull out a section or slice of an array, the colon operator : is used when calling the index. The general form is:

```
<slice> = <array>[start:stop]
```

Where is the slice or section of the array object . The index of the slice is specified in [start:stop]. Remember Python counting starts at 0 and ends at n-1. The index [0:2] pulls the first two values out of an array. The index [1:3] pulls the second and third values out of an array.

An example of slicing the first two elements out of an array is below.

In [18]:

```python
a = np.array([2,4,6])

b = a[0:2]
print(b)
```

```
[2 4]
```

On either sides of the colon, a blank stands for "default".

- [:2] corresponds to [start=default:stop=2]
- [1:] corresponds to [start=1:stop=default] Therefore, the slicing operation [:2] pulls out the first and second values in an array. The slicing operation [1:] pull out the second through the last values in an array.

The example below illustrates the default stop value is the last value in the array.

In [19]:
```
a = np.array([2, 4, 6, 8])
print(a)
b = a[1:]
print(b)
```

```
[2 4 6 8]
[4 6 8]
```

The next examples shows the default start value is the first value in the array.

In [20]:
```
a = np.array([2, 4, 6, 8])
print(a)
b = a[:3]
print(b)
```

```
[2 4 6 8]
[2 4 6]
```

The following indexing operations output the same array.

In [21]:

```
a = np.array([2, 4, 6, 8])
b = a[0:4]
print(b)
c = a[:4]
print(c)
d = a[0:]
print(d)
e = a[:]
print(e)
```

```
[2 4 6 8]
[2 4 6 8]
[2 4 6 8]
[2 4 6 8]
```

## Slicing 2D Arrays

2D NumPy arrays can be sliced with the general form:

```
<slice> = <array>[start_row:end_row, start_col:end_col]
```

The code section below creates a two row by four column array and indexes out the first two rows and the first three columns.

In [22]:

```
a = np.array([[2, 4, 6, 8], [10, 20, 30, 40]])
print(a)
b = a[0:2, 0:3]
print(b)
```

```
[[ 2  4  6  8]
 [10 20 30 40]]
[[ 2  4  6]
 [10 20 30]]
```

The code section below slices out the first two rows and all columns from array a.

In [23]:
```
a = np.array([[2, 4, 6, 8], [10, 20, 30, 40]])
b = a[:2, :]  #[first two rows, all columns]
print(b)
```

```
[[ 2  4  6  8]
 [10 20 30 40]]
```

### * Boolean Indexing

In [24]:
```
bool_index = np.arange(32).reshape(4,8)
print(bool_index)
```

```
[[ 0  1  2  3  4  5  6  7]
 [ 8  9 10 11 12 13 14 15]
 [16 17 18 19 20 21 22 23]
 [24 25 26 27 28 29 30 31]]
```

In [25]:
```
bool_index < 20 # boolean expression
```

Out[25]:
```
array([[ True,  True,  True,  True,  True,  True,  True,  True],
       [ True,  True,  True,  True,  True,  True,  True,  True],
       [ True,  True,  True,  True, False, False, False, False],
       [False, False, False, False, False, False, False, False]])
```

In [26]:
```
'''
The often we use such operaton when we do thresholding on the data. We can use these
operation as an index and can get the result based on the expression. Let's see some
of the examples.
'''
bool_index[ bool_index < 20 ] # only get the values which is less than 20
```

Out[26]:
```
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16,
       17, 18, 19])
```

In [27]:

```
'''
let's see some various example of using this.
'''
print(bool_index[ bool_index % 2 == 0 ])      # [ 0  2  4 ... 28 30]
print(bool_index[ bool_index % 2 != 0 ])      # [ 1  3  5  ... 29 31]
print(bool_index[ bool_index % 2 == 0 ] + 1)  # [ 1  3  5  ... 29 31]
```

```
[ 0  2  4  6  8 10 12 14 16 18 20 22 24 26 28 30]
[ 1  3  5  7  9 11 13 15 17 19 21 23 25 27 29 31]
[ 1  3  5  7  9 11 13 15 17 19 21 23 25 27 29 31]
```

#### Math and Statistical Operation

Basic mathematical functions operate elementwise on arrays.

In [28]:
```
x = np.arange(0,4).reshape(2,2).astype('float64')
y = np.arange(5,9).reshape(2,2).astype('float64')

# Elementwise sum; both produce the array
print(x + y)
print(np.add(x, y))

print('----------------------------')

# Elementwise difference; both produce the array
print(x - y)
print(np.subtract(x, y))

print('----------------------------')

# Elementwise product; both produce the array
print(x * y)
print(np.multiply(x, y))

print('----------------------------')

# Elementwise division; both produce the array
print(x / y)
print(np.divide(x, y))

print('----------------------------')

# Elementwise square root; produces the array
print(np.sqrt(x))
```

```
[[ 5.  7.]
 [ 9. 11.]]
[[ 5.  7.]
 [ 9. 11.]]
----------------------------
[[-5. -5.]
 [-5. -5.]]
[[-5. -5.]
 [-5. -5.]]
----------------------------
```

```
[[ 0.  6.]
 [14. 24.]]
[[ 0.  6.]
 [14. 24.]]
----------------------------
[[0.         0.16666667]
 [0.28571429 0.375     ]]
[[0.         0.16666667]
 [0.28571429 0.375     ]]
----------------------------
[[0.         1.        ]
 [1.41421356 1.73205081]]
```

In [29]:

```
'''
Numpy provides many useful functions for performing computations on arrays;
one of the most useful is sum
'''


x = np.arange(5,9).reshape(2,2).astype('int64')

print(x)
print(np.sum(x))          # Compute sum of all elements; prints "26"
print(np.sum(x, axis=0))  # Compute sum of each column; prints "[12 14]"
print(np.sum(x, axis=1))  # Compute sum of each row; prints "[11 15]"
```

```
[[5 6]
 [7 8]]
26
[12 14]
[11 15]
```

In [30]:

```
'''
Sometimes we need to manipulate the data in array. It can be done by reshaping or transpose
the array. Transposing is a special form of reshaping that similarly returns a view on the
underlying data without copying anything.

When doing matrix computation, we may do this very often.
'''

arr = np.arange(10).reshape(2,5)
print('At first \n', arr)   # At first
                            #  [[0 1 2 3 4]
                            #  [5 6 7 8 9]]
print()
print('After transpose \n', arr.T)


print('----------------------------')
transpose = np.arange(10).reshape(2,5)
print(np.dot(transpose.T, transpose))
```

```
At first
 [[0 1 2 3 4]
 [5 6 7 8 9]]

After transpose
 [[0 5]
 [1 6]
 [2 7]
 [3 8]
 [4 9]]
----------------------------
[[25 30 35 40 45]
 [30 37 44 51 58]
 [35 44 53 62 71]
 [40 51 62 73 84]
 [45 58 71 84 97]]
```

In [31]:

```
'''
statistical functions and concern used function, such as

- mean
- min
- sum
- std
- median
- argmin, argmax
'''

ary = 10 * np.random.randn(2,5)
print('Mean   : ', np.mean(ary))     # 0.9414738037734729
print('STD    : ', np.std(ary))      # 5.897885490589387
print('Median : ', np.median(ary))   # 1.5337461352996276
print('Argmin : ', np.argmin(ary))   # 3
print('Argmax : ', np.argmax(ary))   # 2
print('Max    : ', np.max(ary))      # 10.399663734487659
print('Min    : ', np.min(ary))      # -9.849839643044087
print('Compute mean by column :', np.mean(ary, axis = 0)) # compute the means by column

print('Compute median by row  : ', np.median(ary, axis = 1)) # compute the medians
```

```
Mean   :  -1.3380400711907203
STD    :  9.43232089326773
Median :  2.0096051084337905
Argmin :  1
Argmax :  2
Max    :  13.534226955063268
Min    :  -17.957162260083123
Compute mean by column : [-0.35782662 -7.3556628   1.20686251  4.12493558 -4.30850902]
Compute median by row  :  [2.39262206 1.62658816]
```

### Universal Functions

A universal functions or ufunc is a special function that performs element-wise operations on the data in ndarrays. Such as fast vectorized wrapper for simple functions that take one or more scalar values and produce one or more scalar results. Many ufuncs are simple element-wise transformation, like sqrt or exp.

In [32]:
```
ary = np.arange(5)

print('Find root of each elements-wise \n', np.sqrt(ary))
print()
print('Find exponential for each element-wise \n', np.exp(ary))

# Find root of each elements-wise
#  [0.         1.         1.41421356 1.73205081 2.        ]

# Find exponential for each element-wise
#  [ 1.          2.71828183  7.3890561  20.08553692 54.59815003]


'''
np.maximum

This computed the element-wise maximum of the elements in two array and returned a single
array as a result.
'''
print()
print('Max values between two array \n', np.maximum(np.sqrt(ary), np.exp(ary)))
print('----------------------------')
print()
# Max values between two array
#  [ 1.          2.71828183  7.3890561  20.08553692 54.59815003]


'''
np.modf

Another unfunc but can return multiple arrays. It returns the fractional and integral parts
of a floating point array.
'''
rem, num = np.modf(np.exp(ary))
print('Floating Number ', np.exp(ary))
print()
print('Remainder       ', rem)
print('Number          ', num)
print('----------------------------')
print()

```

```
# Floating Number  [ 1.          2.71828183  7.3890561  20.08553692 54.59815003]

# Remainder        [0.          0.71828183 0.3890561  0.08553692 0.59815003]
# Number           [ 1.  2.  7. 20. 54.]



'''
np.ceil

Return the ceiling of the input, element-wise.
'''
ceil_num = np.array([-1.7, -1.5, -0.2, 0.2, 1.5, 1.7, 2.0])
print(ceil_num)               # [-1.7 -1.5 -0.2  0.2  1.5  1.7  2. ]
print(np.ceil(ceil_num))      # [-1. -1. -0.  1.  2.  2.  2.]


''' not ufunc

np.around

Evenly round to the given number of decimals.
'''
print(np.around(ceil_num))    # [-2. -2. -0.  0.  2.  2.  2.]
print('----------------------------')
print()

'''
np.absolute | np.abs | np.fabs

Calculate the absolute value element-wise.
'''
absl = np.array([-1, 2])
print('Absolute of Real Values              ', np.abs(absl))
print('Absolute of Real Values with Float   ', np.fabs(absl))
print('Absolute of Complex Values          ', np.abs(1.2 + 1j))

# Absolute of Real Values               [1 2]
# Absolute of Real Values with Float    [1. 2.]
# Absolute of Complex Values           1.5620499351813308

```

```
Find root of each elements-wise
 [0.         1.         1.41421356 1.73205081 2.        ]

Find exponential for each element-wise
 [ 1.          2.71828183  7.3890561  20.08553692 54.59815003]

Max values between two array
 [ 1.          2.71828183  7.3890561  20.08553692 54.59815003]
----------------------------

Floating Number  [ 1.          2.71828183  7.3890561  20.08553692 54.59815003]

Remainder        [0.         0.71828183 0.3890561  0.08553692 0.59815003]
Number           [ 1.  2.  7. 20. 54.]
----------------------------

[-1.7 -1.5 -0.2  0.2  1.5  1.7  2. ]
[-1. -1. -0.  1.  2.  2.  2.]
[-2. -2. -0.  0.  2.  2.  2.]
----------------------------

Absolute of Real Values               [1 2]
Absolute of Real Values with Float    [1. 2.]
Absolute of Complex Values           1.5620499351813308
```

### NumPy Random

- np.random.rand
- np.random.randn
- np.random.random
- np.random.random_sample
- np.random.randint
- np.random.normal
- np.ranodm.uniform
- np.random.seed
- np.random.shuffle
- np.random.choice

In [33]:
```
'''
np.random.rand()

Create an array of the given shape and populate it with
random samples from a uniform distribution
over ``[0, 1)``
'''
ary = np.random.rand(5,2) # shape: 5 row, 2 column
print('np.random.rand() \n', ary)
print('---------------------------')
print()

'''
np.random.randn

Return a sample (or samples) from the "standard normal" distribution.
'''
ary = np.random.randn(6)
print('1D array: np.random.randn() \n', ary)
ary = np.random.randn(3,3)
print('2D array: np.random.randn() \n', ary)
print('---------------------------')
print()

'''
np.random.random.
numpy.random.random() is actually an alias for numpy.random.random_sample()

Return a sample (or samples) from the "standard normal" distribution.
'''
ary = np.random.random((3,3))
print('np.random.randn() \n', ary)
ary = np.random.random_sample((3,3))
print('np.random.random_sample() \n', ary)
print('---------------------------')
print()

'''
np.random.randint.
Return random integers from low (inclusive) to high (exclusive)

```

```
Return random integers from the “discrete uniform” distribution of the specified
dtype in the “half-open” interval [low, high). If high is None (the default),
then results are from [0, low).

'''
ary = np.random.randint(low = 2, high = 6, size = (5,5))
print('np.random.randint() \n', ary)
ary = np.random.randint(low = 2, high = 6)
print('np.random.randint() :', ary)

```

```
np.random.rand()
 [[0.97086979 0.4035386 ]
 [0.50018503 0.54193609]
 [0.20686171 0.57199175]
 [0.26977469 0.33028575]
 [0.12754148 0.27597959]]
----------------------------

1D array: np.random.randn()
 [ 0.30039809  0.15921004  0.39210374  0.87442049 -0.48676795 -0.46489751]
2D array: np.random.randn()
 [[-0.71335337 -0.69846494  0.56381866]
 [ 0.38608365 -1.0354181   0.78433437]
 [ 0.04962596 -0.04290496 -0.13177029]]
----------------------------

np.random.randn()
 [[0.71070044 0.97176596 0.36264464]
 [0.57971045 0.94351494 0.31316495]
 [0.53836635 0.98105565 0.11923478]]
np.random.random_sample()
 [[0.02271442 0.54517504 0.46636122]
 [0.79337943 0.01069799 0.828379  ]
 [0.62348799 0.32117798 0.86282151]]
----------------------------

np.random.randint()
 [[5 5 4 5 3]
 [3 2 4 5 3]
 [5 3 5 2 5]
```

```
 [5 4 3 2 4]
 [3 5 2 5 5]]
np.random.randint() : 4
```

**Note: np.random.rand() vs np.random.random_samples()**

Both functions generate samples from the uniform distribution on 0, 1). The only difference is in how the arguments are handled. With numpy.random.rand, the length of each dimension of the output array is a separate argument. With numpy.random.random_sample, the shape argument is a single tuple.

In [34]:

```
'''
np.random.normal()

Draw random samples from a normal (Gaussian) distribution. This is Distribution is
also known as Bell Curve because of its characteristics shape.
'''
mu, sigma = 0, 0.1 # mean and standard deviation
print('np.random.normal() \n',np.random.normal(mu, sigma, 10)) # from doc
print('----------------------------')
print()

'''
np.random.uniform()

Draw samples from a uniform distribution
'''
print('np.random.uniform() \n', np.random.uniform(-1,0,10))
print('----------------------------')
print()

'''
np.random.seed()
'''
np.random.seed(3) # seed the result

'''
np.random.shuffle

Modify a sequence in-place by shuffling its contents
'''

ary = np.arange(9).reshape((3, 3))
print('Before Shuffling \n', ary)

np.random.shuffle(ary)
print('After Shuffling \n', ary)
print('----------------------------')
print()

'''
np.random.choice
```

```

Generates a random sample from a given 1-D array
'''
ary = np.random.choice(5, 3) # Generate a uniform random sample from np.arange(5) of size 3:
print('np.random.choice() \n', ary) # This is equivalent to np.random.randint(0,5,3)

```

```
np.random.normal()
 [-0.04054043  0.05868212 -0.06168353 -0.02128441 -0.04459205 -0.24341877
 -0.04998943  0.05465797  0.01635333 -0.05197681]
----------------------------

np.random.uniform()
 [-0.49809706 -0.83044643 -0.28303285 -0.52032418 -0.90501339 -0.17021207
 -0.32933149 -0.08523018 -0.90810189 -0.43904442]
----------------------------

Before Shuffling
 [[0 1 2]
 [3 4 5]
 [6 7 8]]
After Shuffling
 [[3 4 5]
 [0 1 2]
 [6 7 8]]
----------------------------

np.random.choice()
 [1 3 0]
```

Some used functions:

- sort()
- unique()
- vstack() and hstack()
- ravel()
- tile()
- concatenate()

In [35]:

```
'''
sort()
'''
# create a 10 element array of randoms
unsorted = np.random.randn(10)
print('Unsorted \n', unsorted)

# inplace sorting
unsorted.sort()
print('Sorted \n', unsorted)

print()
print('----------------------------------')


'''
unique()
'''
ary = np.array([1,2,1,4,2,1,4,2])
print('Unique values : ', np.unique(ary))
print()
print('----------------------------------')


'''
vstack and hstack
'''
arx = np.array([[1,2,3],[3,4,5]])
ary = np.array([[4,5,6],[7,8,9]])
print('Vertical Stack \n', np.vstack((arx,ary)))
print('Horizontal Stack \n', np.hstack((arx,ary)))
print('Concate along columns \n', np.concatenate([arx, ary], axis = 0)) # similar vstack
print('Concate along rows \n', np.concatenate([arx, ary], axis = 1))    # similar hstack
print()
print('----------------------------------')


'''
ravel :  convert one numpy array into a single column
'''
ary = np.array([[1,2,3],[3,4,5]])
```

```
print('Ravel \n', ary.ravel())
print()
print('---------------------------------')


'''
tile()
'''
ary = np.array([-1, 0, 1])
ary_tile = np.tile(ary, (4, 1)) # Stack 4 copies of v on top of each other
print('tile array \n', ary_tile)

```

```
Unsorted
 [ 0.1841282  -1.00595517 -0.34198034 -0.04472413  0.27844092 -0.58089402
 -0.15151488 -1.14743417 -0.61100002 -1.18951737]
Sorted
 [-1.18951737 -1.14743417 -1.00595517 -0.61100002 -0.58089402 -0.34198034
 -0.15151488 -0.04472413  0.1841282   0.27844092]

---------------------------------
Unique values :  [1 2 4]

---------------------------------
Vertical Stack
 [[1 2 3]
 [3 4 5]
 [4 5 6]
 [7 8 9]]
Horizontal Stack
 [[1 2 3 4 5 6]
 [3 4 5 7 8 9]]
Concate along columns
 [[1 2 3]
 [3 4 5]
 [4 5 6]
 [7 8 9]]
Concate along rows
 [[1 2 3 4 5 6]
 [3 4 5 7 8 9]]
```

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

----------------------------------
tile array
 [[-1  0  1]
 [-1  0  1]
 [-1  0  1]
 [-1  0  1]]
```

In [36]:

```
# Set Function

s1 = np.array(['desk','chair','bulb'])
s2 = np.array(['lamp','bulb','chair'])
print(s1, s2)

print( np.intersect1d(s1, s2) )
print( np.union1d(s1, s2) )
print( np.setdiff1d(s1, s2) )# elements in s1 that are not in s2
print( np.in1d(s1, s2) )
```

```
['desk' 'chair' 'bulb'] ['lamp' 'bulb' 'chair']
['bulb' 'chair']
['bulb' 'chair' 'desk' 'lamp']
['desk']
[False  True  True]
```

### Broadcasting

Broadcasting is a powerful mechanism that allows numpy to work with arrays of different shapes when performing arithmetic operations. Frequently we have a smaller array and a larger array, and we want to use the smaller array multiple times to perform some operation on the larger array

In [37]:

```
start = np.zeros((4,3))
print(start)
print('----------------------------------')
# create a rank 1 ndarray with 3 values
add_rows = np.array([1, 0, 2])

y = start + add_rows  # add to each row of 'start' using broadcasting
print(y)
print('----------------------------------')
# create an ndarray which is 4 x 1 to broadcast across columns
add_cols = np.array([[0,1,2,3]])
add_cols = add_cols.T

print(add_cols)
print('----------------------------------')

# this will just broadcast in both dimensions
add_scalar = np.array([1])
print(start + add_scalar)
```

```
[[0. 0. 0.]
 [0. 0. 0.]
 [0. 0. 0.]
 [0. 0. 0.]]
----------------------------------
[[1. 0. 2.]
 [1. 0. 2.]
 [1. 0. 2.]
 [1. 0. 2.]]
----------------------------------
[[0]
 [1]
 [2]
 [3]]
----------------------------------
[[1. 1. 1.]
 [1. 1. 1.]
 [1. 1. 1.]
 [1. 1. 1.]]
```