# Introduction to NumPy

- Author: Ronisson Lucas Calmon da Conceição ## Introduction

NumPy is a library for handling multidimensional arrays, with several applications in numerical calculus, linear algebra, machine learning and several other functions for scientific programming. Its implementation is based on the C language, which provides high performance in the treatment and manipulation of arrays , through various functions and operators. This notebook aims to present a brief introduction to NumPy. Functions applicable to linear algebra will be dealt with separately in another notebook.

## Installation

Initially, it is necessary to install the NumPy library. To do this, Windows users will be able to run the command prompt and enter the command below.

*pip install numpy*

For Windows users, the installation can be done by Anaconda as well:

*conda install numpy*

For Linux users (Ubuntu and Debian):

*sudo apt-get install python-numpy*

## Importing NumPy

The first step is to import the NumPy library, for that we can use the command below:

In [1]:

```
import numpy as np
```

To check the running NumPy version the following command can be used:

In [2]:

```
np.__version__
```

Out[2]:

```
'1.19.5'
```

# ndarray object

The primary object when working with Numpy is the ndarray (class). In other words, when creating an array with this library, we are instantiating objects of the ndarray class, which inherit a set of methods and attributes that we will see in this notebook.

We can create an ndarray object with the np.array() function - in addition to several others that we'll see later. For that, we pass a list or iterable containing the respective elements as argument of this function. Let's look at this process in the line of code below.

In [3]:

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

To check if the created object is in fact an ndarray, we pass the name of the created array into the built-in type() function:

In [4]:

```
type(b)
```

Out[4]:

```
numpy.ndarray
```

We can check the data type associated with the created object through the *dtype* attribute.

In [5]:

```
b.dtype
```

Out[5]:

```
dtype('int64')
```

To check the number of dimensions of the created array we use the *ndim* attribute. The *size* attribute shows the size of the array, while the shape attribute shows the number of elements in each dimension of the array.

In [6]:

```
b.ndim
```

Out[6]:

```
1
```

In [7]:

```
b.shape
```

Out[7]:

```
(5,)
```

In [8]:

```
b.size
```

Out[8]:

```
5
```

Array b is one-dimensional. We can create an array of two or more dimensions (as we will see later). Let's look at an example:

In [9]:

```
k = np.array([[10.5, 1.3, 2], [4.3, 5.2, 1.8]])
print('Shape: ', k.shape)
print('Number of dimensions: ', k.ndim)
print('Size: ', k.size)
print('Dtype: ', k.dtype)
```

```
Shape:  (2, 3)
Number of dimensions:  2
Size:  6
Dtype:  float64
```

To check the size in bytes of each element we use the *itemsize* attribute.

In [10]:

```
k.itemsize
```

Out[10]:

```
8
```

In summary, the array object has the following main attributes:

| Attribute | Description |
|---|---|
| shape | Tuple that contains the number of elements for each dimension of the array. |
| size | Total number of elements in an array. |
| ndim | Number of dimensions. |
| nbytes | Number of bytes used to store the data. |
| dtype | Data type of elements present in an array. |

# Data Types

Earlier, we saw that the dtype attribute provides the data type of elements in a given array.

In [11]:

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

Out[11]:

```
dtype('int64')
```

Another possibility would be to create an array with strings.

In [12]:

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

Out[12]:

```
dtype('<U1')
```

We can change the data type of an array by passing the dtype argument into the np.array() function. We check that array *a* has elements of type int. Let's look at how to explicitly specify another data type when creating an array.

In [13]:

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

Out[13]:

```
dtype('float64')
```

In this case, our new *a* array has the same numbers as the previous example, despite its data type being float. Finally, the *d* array has elements of type *complex*.

In [14]:

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

Out[14]:

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

## astype

After creating an ndarray we can convert its data type with the astype function:

In [15]:

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

Out[15]:

```
dtype('int64')
```

In [16]:

```
z = z.astype(str, copy=True)

z.dtype
```

Out[16]:

```
dtype('<U21')
```

## Creating different types of arrays

We will learn in this subsection how to use different NumPy functions to create arrays.

We will create a 2-dimensional array by passing nested lists as argument to the np.array() function:

In [17]:

```
array1 = np.array([[4,5,6],[7,8,9],[1,0,1]])
array1
```

Out[17]:

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

In addition to lists, we can pass tuples as an argument to the np.array() function.

In [18]:

```
array2 = np.array(((4,5,6),(7,8,9),(1,0,1)))
array2
```

Out[18]:

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

We can also simultaneously use lists and tuples as arguments:

In [19]:

```
array3 = np.array([(1,2,3),[4,5,6]])
array3
```

Out[19]:

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

The np.zeros() function allows you to create an array containing 0 in all its entries. To do so, we pass the *shape* of the array as an argument to the function.

In [20]:

```
zero = np.zeros(shape = (2,2))
zero
```

Out[20]:

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

The np.ones() function creates an array containing 1 in all of its entries. Its implementation is similar to the previous function.

In [21]:

```
i = np.ones(shape = (2,2))
i
```

Out[21]:

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

The default data type of this function is float. However, we can pass the dtype argument inside the *np.ones()* function to change it to another type.

In [22]:

```
i.dtype
```

Out[22]:

```
dtype('float64')
```

In [23]:

```
i = np.ones(shape = (2,2), dtype=int)
i.dtype
```

Out[23]:

```
dtype('int64')
```

With the *np.arange()* function we can create a sequence of numbers, similar to the built-in *range()* function for integers.

In the example below we create a sequence of integers between 2010 and 2021. But, it should be noted that the first to last number is $n - 1$. So, we can think of this function from the following syntax: np.arange(start, end, step). The last argument (step) denotes the spacing between values.

In [24]:

```
np.arange(2010,2021)
```

Out[24]:

```
array([2010, 2011, 2012, 2013, 2014, 2015, 2016, 2017, 2018, 2019,
2020])
```

If we don't specify a start value, the default value will be 0.

In [25]:

```
np.arange(10)
```

Out[25]:

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

The example below creates a sequence of numbers from 1 to 10 with a 2 spacing (step).

In [26]:

```
np.arange(1,11,2)
```

Out[26]:

```
array([1, 3, 5, 7, 9])
```

To invert a sequence of numbers inside np.arange() we can pass -1 as step.

In [27]:

```
np.arange(2020,2009,-1)
```

Out[27]:

```
array([2020, 2019, 2018, 2017, 2016, 2015, 2014, 2013, 2012, 2011,
2010])
```

In this example, we create a sequence of numbers from 2010 to 2021 of 5 out of 5.

In [28]:

```
np.arange(2010,2021,5)
```

Out[28]:

```
array([2010, 2015, 2020])
```

We can use np.arange() with the reshape function to create a two-dimensional array, that is, transform a one-dimensional array into an array.

In [29]:

```
np.arange(1,31)
```

Out[29]:

```
array([ 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])
```

In [30]:

```
np.arange(1,31).reshape(3,10)
```

Out[30]:

```
array([[ 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]])
```

Or yet,

In [31]:

```
np.arange(1,31).reshape(10,3)
```

Out[31]:

```
array([[ 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]])
```

In [32]:

```
np.arange(1,17).reshape(4,4)
```

Out[32]:

```
array([[ 1,  2,  3,  4],
       [ 5,  6,  7,  8],
       [ 9, 10, 11, 12],
       [13, 14, 15, 16]])
```

The np.linspace() function returns an array of evenly spaced numbers within a defined range.

Let's create an array with numbers within the range [1, 21] in 5 parts:

In [33]:

```
np.linspace(1,21,5)
```

Out[33]:

```
array([ 1.,  6., 11., 16., 21.])
```

NumPy has several functions to generate random numbers within the np.random module. We will generate 10 numbers random values between 0 and 1 with this module's random() function.

In [34]:

```
A = np.random.random(10)
A
```

Out[34]:

```
array([0.14180973, 0.23176465, 0.88446946, 0.69729558, 0.01416261,
       0.24825137, 0.41439332, 0.54898777, 0.85208243, 0.87713068])
```

We can transform this one-dimensional vector into an array with the reshape function.

In [35]:

```
A = A.reshape(5,2)
A
```

Out[35]:

```
array([[0.14180973, 0.23176465],
       [0.88446946, 0.69729558],
       [0.01416261, 0.24825137],
       [0.41439332, 0.54898777],
       [0.85208243, 0.87713068]])
```

With the np.full() function we can create arrays that will be fully filled with a given value.

In [36]:

```
np.full(shape = 10, fill_value = 3)
```

Out[36]:

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

In [37]:

```
np.full(shape = (5,5),fill_value = 10)
```

Out[37]:

```
array([[10, 10, 10, 10, 10],
       [10, 10, 10, 10, 10],
       [10, 10, 10, 10, 10],
       [10, 10, 10, 10, 10],
       [10, 10, 10, 10, 10]])
```

We can also create an identity function, with the *np.eye()* or *np.identity()* function. We pass the order of the identity matrix as an argument.

In [38]:

```
np.eye(3)
```

Out[38]:

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

In [39]:

```
np.identity(3)
```

Out[39]:

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

# Operations

In this subsection we will learn how to perform arithmetic operations between arrays.

We'll define two arrays and then perform arithmetic operations, element by element. We will return another array containing the result of the specified operation.

In [40]:

```
a = np.array([10, 15, 20])
b = np.array([2, 5, 7])
```

We can add 2 to each number in the array *a*:

In [41]:

```
a+2
```

Out[41]:

```
array([12, 17, 22])
```

Or subtract 1 from each element of *b*:

In [42]:

```
b-1
```

Out[42]:

```
array([1, 4, 6])
```

Multiply or divide by scalar:

In [43]:

```
b*2
```

Out[43]:

```
array([ 4, 10, 14])
```

In [44]:

```
b/2
```

Out[44]:

```
array([1. , 2.5, 3.5])
```

Get the integer division of an array by a scalar:

In [45]:

```
b//2
```

Out[45]:

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

Let's see how to perform element-by-element operations with *a* and *b*:

In [46]:

```
a+b
```

Out[46]:

```
array([12, 20, 27])
```

In [47]:

```
a-b
```

Out[47]:

```
array([ 8, 10, 13])
```

In [48]:

```
a*b
```

Out[48]:

```
array([ 20,  75, 140])
```

In [49]:

```
a/b
```

Out[49]:

```
array([5.        , 3.        , 2.85714286])
```

We could have used the *np.add()*, *np.subtract()*, *np.multiply()* and *np.divide()* functions to get the same results as above.

In [50]:

```
np.add(a,b)
```

Out[50]:

```
array([12, 20, 27])
```

In [51]:

```
np.subtract(a,b)
```

Out[51]:

```
array([ 8, 10, 13])
```

In [52]:

```
np.multiply(a,b)
```

Out[52]:

```
array([ 20,  75, 140])
```

In [53]:

```
np.divide(a,b)
```

Out[53]:

```
array([5.        , 3.        , 2.85714286])
```

Let's square each element of *c* with the *np.power()* function. Next, we will obtain the square root of each element of this vector with the *np.sqrt()* function.

In [54]:

```
c = np.array([4,16, 25])
```

In [55]:

```
np.power(c, 2)
```

Out[55]:

```
array([ 16, 256, 625])
```

In [56]:

```
np.sqrt(c)
```

Out[56]:

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

Multiplying *a* array elements by the square root of *c* elements

In [57]:

```
np.multiply(a,np.sqrt(c))
```

Out[57]:

```
array([ 20.,  60., 100.])
```

For multidimensional arrays the operations defined here continue to be performed element by element.

In [58]:

```
a = np.full(shape = (3,3),fill_value = 5)
b = np.ones(shape = (3,3))
```

In [59]:

```
a
```

Out[59]:

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

In [60]:

```
b
```

Out[60]:

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

In [61]:

```
np.add(a,b)
```

Out[61]:

```
array([[6., 6., 6.],
       [6., 6., 6.],
       [6., 6., 6.]])
```

In [62]:

```
np.subtract(a,b)
```

Out[62]:

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

In [63]:

```
np.divide(b,a)
```

Out[63]:

```
array([[0.2, 0.2, 0.2],
       [0.2, 0.2, 0.2],
       [0.2, 0.2, 0.2]])
```

In [64]:

```
np.multiply(a,b)
```

Out[64]:

```
array([[5., 5., 5.],
       [5., 5., 5.],
       [5., 5., 5.]])
```

In [65]:

```
a*b
```

Out[65]:

```
array([[5., 5., 5.],
       [5., 5., 5.],
       [5., 5., 5.]])
```

Thus, the * operator does not effect the matrix product. We will see later how to use this library to calculate the product between matrices.

## Comparation

The operators >, >=, <=, ==, != perform element-by-element comparison between arrays.

In [66]:

```
a = np.ones(5)
b = np.full(shape=5,fill_value=1)
c = np.arange(5)

a
```

Out[66]:

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

In [67]:

```
b
```

Out[67]:

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

In [68]:

```
c
```

Out[68]:

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

In [69]:

```
a == c  # checks for equality between element-by-element arrays
```

Out[69]:

```
array([False,  True, False, False, False])
```

In [70]:

```
a != c  # check element-by-element difference between arrays
```

Out[70]:

```
array([ True, False,  True,  True,  True])
```

For the other operators we will have the same validation (element by element):

In [71]:

```
c > a
```

Out[71]:

```
array([False, False,  True,  True,  True])
```

In [72]:

```
c > 2  # comparison with a scalar
```

Out[72]:

```
array([False, False, False,  True,  True])
```

In [73]:

```
a < c
```

Out[73]:

```
array([False, False,  True,  True,  True])
```

In [74]:

```
c >= a
```

Out[74]:

```
array([False,  True,  True,  True,  True])
```

In [75]:

```
c <= a
```

Out[75]:

```
array([ True,  True, False, False, False])
```

## Increment and Decrement

There are no ++ or -- specific operators to perform increment or decrement operations. For this, we can carry out a reallocation of values. For increment: array = array + scalar or array+ = scalar. For -, *, /, we use analog syntax.

In [76]:

```
ar = np.ones(10)
ar
```

Out[76]:

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

In [77]:

```
ar = ar+1 # increment each value with unit

ar
```

Out[77]:

```
array([2., 2., 2., 2., 2., 2., 2., 2., 2., 2.])
```

In [78]:

```
ar = ar-1 # decrements each value by one unit

ar
```

Out[78]:

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

In [79]:

```
ar+=5 # equals ar = ar + 5

ar
```

Out[79]:

```
array([6., 6., 6., 6., 6., 6., 6., 6., 6., 6.])
```

In [80]:

```
ar-= 2 # equals ar = ar - 2

ar
```

Out[80]:

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

In [81]:

```
ar/= 2 # or ar = ar/2

ar
```

Out[81]:

```
array([2., 2., 2., 2., 2., 2., 2., 2., 2., 2.])
```

In [82]:

```
ar*=5  # or ar = ar*5

ar
```

Out[82]:

```
array([10., 10., 10., 10., 10., 10., 10., 10., 10., 10.])
```

## Manipulating the shape

NumPy makes it possible to modify the shape of an array after its creation. There are two possibilities for implementation. We will transform a one-dimensional array with shape $(16,)$ into a two-dimensional array with shape $(4,4)$.

In [83]:

```
w = np.arange(16)

w.shape
```

Out[83]:

```
(16,)
```

In [84]:

```
w.shape = (4,4)

w
```

Out[84]:

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

With the reshape function we can do an operation analogous to the previous one, passing a tuple with the new shape of the array.

In [85]:

```
a = np.arange(9)
a = a.reshape((3,3))

a
```

Out[85]:

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

We will use the *ravel()* function to perform the inverse operation from the $a$ array, converting a 2-D (two-dimensional) array into 1-D.

In [86]:

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

Out[86]:

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

Another possible manipulation is to get the transpose of an array. This operation is not valid for 1-D arrays.

In [87]:

```
w.shape = (16,)
w.shape
```

Out[87]:

```
(16,)
```

In [88]:

```
w = w.T
w.shape
```

Out[88]:

```
(16,)
```

For a matrix we have to:

In [89]:

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

Out[89]:

```
(3, 2)
```

In [90]:

```
a = a.transpose()  # or: a = a.T

a.shape
```

Out[90]:

```
(2, 3)
```

## Indexing and Slicing

In this section we will learn how to select element through indexing and slicing.

Indexing NumPy arrays is similar to Python lists, in that each element occupies a certain index. In the array, the number 10 occupies index 0, 15 has index 1, and so on.

In [91]:

```
a = np.arange(10,30,5)
a
```

Out[91]:

```
array([10, 15, 20, 25])
```

So, to access a certain element we use *array[index]*. So, to access the number 10, which has index 0:

In [92]:

```
a[0]
```

Out[92]:

```
10
```

For the other elements:

In [93]:

```
a[1]
```

Out[93]:

```
15
```

In [94]:

```
a[2]
```

Out[94]:

```
20
```

NumPy arrays also support negative indexing, when we start counting from the last element that has index -1 to the first element of the array. In our example, the number 25 has index -1, while the number 20 has index -2, successively up to element 10 that has index -4.

In [95]:

```
a[-1]
```

Out[95]:

```
25
```

In [96]:

```
a[-2]
```

Out[96]:

```
20
```

In [97]:

```
a[-4]
```

Out[97]:

```
10
```

Accessing multiple elements through the index:

In [98]:

```
a[[1,3]]
```

Out[98]:

```
array([15, 25])
```

2-D arrays

- rectangular structure of representation in rows and columns
- defined by two axes, where 0 corresponds to rows 1 to columns
- indexing is a value pair (row,column)
- array[row,column]

In [99]:

```
matrix = np.arange(50,65).reshape(3,5)
matrix
```

Out[99]:

```
array([[50, 51, 52, 53, 54],
       [55, 56, 57, 58, 59],
       [60, 61, 62, 63, 64]])
```

To access number 57:

In [100]:

```
matrix[1,2]
```

Out[100]:

```
57
```

slicing

- Extract parts of an array or generate new ones

In [101]:

```
array = np.arange(30,50)
array
```

Out[101]:

```
array([30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44,
45, 46,
       47, 48, 49])
```

In [102]:

```
array[:]
```

Out[102]:

```
array([30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44,
45, 46,
       47, 48, 49])
```

In [103]:

```
array[1:]
```

Out[103]:

```
array([31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45,
46, 47,
       48, 49])
```

In [104]:

```
array[:10]
```

Out[104]:

```
array([30, 31, 32, 33, 34, 35, 36, 37, 38, 39])
```

In [105]:

```
array[1:10]
```

Out[105]:

```
array([31, 32, 33, 34, 35, 36, 37, 38, 39])
```

In [106]:

```
array[::2]
```

Out[106]:

```
array([30, 32, 34, 36, 38, 40, 42, 44, 46, 48])
```

In [107]:

```
array[5:15:2]
```

Out[107]:

```
array([35, 37, 39, 41, 43])
```

In [108]:

```
array[:10:5]
```

Out[108]:

```
array([30, 35])
```

In [109]:

```
array[::]
```

Out[109]:

```
array([30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44,
45, 46,
       47, 48, 49])
```

matrix[row,column]

In [110]:

```
matrix = np.arange(10,30).reshape(10,2)
matrix
```

Out[110]:

```
array([[10, 11],
       [12, 13],
       [14, 15],
       [16, 17],
       [18, 19],
       [20, 21],
       [22, 23],
       [24, 25],
       [26, 27],
       [28, 29]])
```

Selecting the first line

In [111]:

```
matrix[0,:]
```

Out[111]:

```
array([10, 11])
```

Selecting the first column

In [112]:

```
matrix[:,0].reshape(-1,1)
```

Out[112]:

```
array([[10],
       [12],
       [14],
       [16],
       [18],
       [20],
       [22],
       [24],
       [26],
       [28]])
```

Selecting the second column

In [113]:

```
matrix[:,1].reshape(-1,1)
```

Out[113]:

```
array([[11],
       [13],
       [15],
       [17],
       [19],
       [21],
       [23],
       [25],
       [27],
       [29]])
```

Extracting a part of the matrix we have:

In [114]:

```
matrix[4:6,:]
```

Out[114]:

```
array([[18, 19],
       [20, 21]])
```

In [115]:

```
matrix[[5,8],:]
```

Out[115]:

```
array([[20, 21],
       [26, 27]])
```

In [116]:

```
matrix[[5,8],1].reshape(-1,1)
```

Out[116]:

```
array([[21],
       [27]])
```

Selecting elements with Boolean variables and conditions

In [117]:

```
b = np.array([
    np.full(3,1),
    np.random.randint(15,20,size=3),
    np.arange(10,13)
])

b
```

Out[117]:

```
array([[ 1,  1,  1],
       [19, 19, 16],
       [10, 11, 12]])
```

In [118]:

```
b<=1
```

Out[118]:

```
array([[ True,  True,  True],
       [False, False, False],
       [False, False, False]])
```

The result is an array with a Boolean value of *True* or *False* in their respective entries, showing whether the specified condition is true or not.

In [119]:

```
b[b<=1]
```

Out[119]:

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

In [120]:

```
b[b>=15].reshape(-1,1)
```

Out[120]:

```
array([[19],
       [19],
       [16]])
```

In [121]:

```
b>15
```

Out[121]:

```
array([[False, False, False],
       [ True,  True,  True],
       [False, False, False]])
```

In [122]:

```
k = np.random.random(size = (5,5))
k
```

Out[122]:

```
array([[0.51185429, 0.0481143 , 0.76116412, 0.28478239, 0.3938322
7],
       [0.56275809, 0.97651424, 0.59940382, 0.03561221, 0.4660367
6],
       [0.59768408, 0.50014782, 0.94917262, 0.97978587, 0.5836902
7],
       [0.39985134, 0.68142565, 0.25063532, 0.81179137, 0.4751119
5],
       [0.54412089, 0.67299579, 0.08442657, 0.32183994, 0.8683314
2]])
```

In [123]:

```
k>0.5
```

Out[123]:

```
array([[ True, False,  True, False, False],
       [ True,  True,  True, False, False],
       [ True,  True,  True,  True,  True],
       [False,  True, False,  True, False],
       [ True,  True, False, False,  True]])
```

In [124]:

```
k[k>0.5]
```

Out[124]:

```
array([0.51185429, 0.76116412, 0.56275809, 0.97651424, 0.59940382,
       0.59768408, 0.50014782, 0.94917262, 0.97978587, 0.58369027,
       0.68142565, 0.81179137, 0.54412089, 0.67299579, 0.86833142])
```

# Iterating over arrays

In [125]:

```
z = np.array([1,2,3,4,5])
for number in z:
    print(number)
```

```
1
2
3
4
5
```

In [126]:

```
matrix = np.array([[1,2,3],[4,5,6],[7,8,9]])
for line in matrix:
    print(line)
```

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

In [127]:

```
for line in matrix:
    for number in line:
        print(number)
```

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

In [128]:

```
for number in matrix.flat:
    print(number)
```

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

In [129]:

```
matrix
```

Out[129]:

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

In [130]:

```
np.apply_along_axis(func1d = np.mean, axis = 0, arr = matrix)
```

Out[130]:

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

In [131]:

```
np.apply_along_axis(func1d = np.sum, axis = 0, arr = matrix)
```

Out[131]:

```
array([12, 15, 18])
```

In [132]:

```
def exp(x):
    return x**2

np.apply_along_axis(func1d = exp, axis = 0, arr = matrix)
```

Out[132]:

```
array([[ 1,  4,  9],
       [16, 25, 36],
       [49, 64, 81]])
```

# Aggregate Functions

Operates on a set of values and outputs a single numeric result.

In [133]:

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

a.sum()
```

Out[133]:

```
6
```

In [134]:

```
a.min()
```

Out[134]:

```
1
```

In [135]:

```
a.max()
```

Out[135]:

```
3
```

In [136]:

```
a.mean()
```

Out[136]:

```
2.0
```

In [137]:

```
a.std()
```

Out[137]:

```
0.816496580927726
```

In [138]:

```
u = np.array([7, 10, 9, 8, 11, 11, 12])

def agreggate_summary(array):
    import numpy as np
    if isinstance(array,np.ndarray):
        print('Sum: ', np.sum(array))
        print('Minimum value: ', np.min(array))
        print('Maximum value: ', np.max(array))
        print('Average: ', np.mean(array).round(2))
        print('Median: ', np.median(array))
        print('Standard deviation: ', np.std(array).round(2))
        pass

    return None

agreggate_summary(u)
```

```
Sum:  68
Minimum value:  7
Maximum value:  12
Average:  9.71
Median:  10.0
Standard deviation:  1.67
```

In [139]:

```
agreggate_summary([1,2,3])
```

# Universal Functions(ufunc)

In [140]:

```
a = np.linspace(1,50,10)
a
```

Out[140]:

```
array([ 1.        ,  6.44444444, 11.88888889, 17.33333333, 22.77777
778,
       28.22222222, 33.66666667, 39.11111111, 44.55555556, 50.
])
```

In [141]:

```
np.sin(a)
```

Out[141]:

```
array([ 0.84147098,  0.16056113, -0.62683289, -0.99851122, -0.70797
672,
        0.05208808,  0.77760805,  0.98741822,  0.54237243, -0.26237
485])
```

In [142]:

```
np.log(a)
```

Out[142]:

```
array([0.        , 1.86321843, 2.47560426, 2.85263143, 3.1257854 ,
       3.34010969, 3.51650823, 3.6664066 , 3.79673685, 3.91202301])
```

In [143]:

```
b = np.array([4, 16, 25, 49])
np.sqrt(b)
```

Out[143]:

```
array([2., 4., 5., 7.])
```

## Joining e splitting

In [144]:

```
a = np.array([[1,2,3,4],[5,6,7,8]])
b = np.full(shape = (2,4), fill_value = 5)
```

In [145]:

```
np.vstack((a,b))
```

Out[145]:

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

In [146]:

```
np.hstack((a,b))
```

Out[146]:

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

In [147]:

```
array1 = np.array([0,1,-1])
array2 = np.array([5,8,11])
array3 = np.array([17, 20, -2])
```

In [148]:

```
np.row_stack((array1, array2, array3))
```

Out[148]:

```
array([[ 0,  1, -1],
       [ 5,  8, 11],
       [17, 20, -2]])
```

In [149]:

```
np.column_stack((array3, array2, array1))
```

Out[149]:

```
array([[17,  5,  0],
       [20,  8,  1],
       [-2, 11, -1]])
```

- Split an array into parts

$C_{6\times10}$

In [150]:

```
C = np.arange(1,61).reshape(6,10)
```

In [151]:

```
C
```

Out[151]:

```
array([[ 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, 32, 33, 34, 35, 36, 37, 38, 39, 40],
       [41, 42, 43, 44, 45, 46, 47, 48, 49, 50],
       [51, 52, 53, 54, 55, 56, 57, 58, 59, 60]])
```

In [152]:

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

Matrix:

- $A_{6\times5} * B_{6\times5}$

In [153]:

```
A.shape
```

Out[153]:

```
(6, 5)
```

In [154]:

```
A
```

Out[154]:

```
array([[ 1,  2,  3,  4,  5],
       [11, 12, 13, 14, 15],
       [21, 22, 23, 24, 25],
       [31, 32, 33, 34, 35],
       [41, 42, 43, 44, 45],
       [51, 52, 53, 54, 55]])
```

In [155]:

```
B
```

Out[155]:

```
array([[ 6,  7,  8,  9, 10],
       [16, 17, 18, 19, 20],
       [26, 27, 28, 29, 30],
       [36, 37, 38, 39, 40],
       [46, 47, 48, 49, 50],
       [56, 57, 58, 59, 60]])
```

In [156]:

```
[E,F] = np.vsplit(C,2)
```

In [157]:

```
E
```

Out[157]:

```
array([[ 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]])
```

In [158]:

```
F
```

Out[158]:

```
array([[31, 32, 33, 34, 35, 36, 37, 38, 39, 40],
       [41, 42, 43, 44, 45, 46, 47, 48, 49, 50],
       [51, 52, 53, 54, 55, 56, 57, 58, 59, 60]])
```

## Linear algebra

In [159]:

```
a = np.array([[1,2],[3,4]])
b = np.array([[5,1],[-1,2]])
```

In [160]:

```
a.dot(b)
```

Out[160]:

```
array([[ 3,  5],
       [11, 11]])
```

In [161]:

```
np.dot(a,b)
```

Out[161]:

```
array([[ 3,  5],
       [11, 11]])
```

The product does not follow commutativity.