# BASIC ARITHMETIC OPERATIONS ON NUMPY ARRAYS

Basic mathematical functions perform element-wise operation on arrays and are available both as operator overloads and as functions in the NumPy module. We create 2 arrays using array and arrange function and perform the arithmetic operations on them.

The operations performed are - add, subtract, multiply, divide, dot, reciprocal, power, real, imag, conj.

## MATHEMATICAL FUNCTIONS IN NUMPY

Trigonometric operations like **sin()**, **cos(),** and **tan()** are supported . The **floor()** function having syntax as floor(x), returns the largest integer value less than or equal to x, element-wise . The **ceil()** function having a syntax as ceil(x), returns the smallest integer value greater than or equal to x, element-wise . The **sqrt()** function returns the positive square-root of an array, element-wise . The **maximum()** function compares two arrays and returns a new array containing element-wise maximum of array elements ­. The **minimum()** Function compares two arrays and returns a new array containing element-wise minimum of array elements. The **sum()** function returns the sum of array elements over a given axis. When axis=0, it returns the sum of columns and when axis=1, it returns the sum of rows. All of the above functions return a new array.

# CHANGING THE SHAPE OF ARRAY

It is possible to change the dimensions of an array using ravel() and reshape() functions. Both *ravel()* and *reshape()* return a modified array but do not change the original array. The function *ravel()* returns a flattened array, such as a one-dimensional array, containing the elements of the input. The function *reshape()* gives a new shape to an array without changing its data.

## STACKING AND SPLITTING OF ARRAYS

You can stack several arrays together or split an array to several arrays. Several arrays can be stacked together along different dimensions using *vstack()* and *hstack()* functions. With *vstack()*  and *hstack()* functions you are stacking the arrays *a* and *b* together in row-wise and column-wise fashion. The number of columns when stacking with *vstack()* and the number of rows when stacking with *hstack()* should be the same. Use hsplit() function to split an array along its horizontal axis. You can either specify the number of equally shaped arrays to return or specify the columns after which the division should occur. Use vsplit() function to split an array along the vertical axis.

## BROADCASTING IN ARRAYS

The term broadcasting describes how NumPy treats arrays with different shapes during arithmetic operations. Broadcasting allows NumPy functions to deal in a meaningful way with input arrays that do not have exactly the same shape. Subject to certain constraints, the smaller array is “broadcast” across the larger array so that they have compatible shapes and occurs automatically whenever possible. The rules of broadcasting are:

• Rule 1 → If two input arrays do not have the same number of dimensions, a “1” will repeatedly be padded to the shape of the smaller array on its left side by NumPy so both the arrays have the same number of dimensions.

• Rule 2 → If the shape of two input arrays does not match, then the array with a shape of “1” along a particular dimension is stretched by NumPy to match the shape of the array having the largest shape along that dimension. The value of the array element is assumed to be the same along that dimension for the “broadcast” array. After application of the broadcasting rules, the sizes of all arrays must match.

• Rule 3 → If the above two rules are not met, a *ValueError: frames are not aligned* exception is thrown, indicating that the arrays have incompatible shapes.

*The Above Rules can be applied to arrays with any number of dimensions.*

*Example-1.*

*1. >>> import numpy as np*

*2. >>> array\_1 = np.ones([4, 5])*

*3. >>> array\_2 = np.arange(5)*

*4. >>> array\_1*

*array([[1., 1., 1., 1., 1.],*

*[1., 1., 1., 1., 1.],*

*[1., 1., 1., 1., 1.],*

*[1., 1., 1., 1., 1.]])*

*5. >>> array\_2*

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

*6. >>> array\_1.shape*

*(4, 5)*

*7. >>> array\_2.shape*

*(5,)*

*8. >>> array\_1 + array\_2*

*array([[1., 2., 3., 4., 5.],*

*[1., 2., 3., 4., 5.],*

*[1., 2., 3., 4., 5.],*

*[1., 2., 3., 4., 5.]])*

*NumPy operations are usually done on pairs of arrays on an element-by-element basis.*

*In the simplest case, two arrays must have exactly the same shape. NumPy’s broadcasting*

*rule relaxes this constraint when the arrays’ shapes meet certain rules. In the above*

*code, two arrays, array\_1 ➁ and array\_2 ➂, with different dimensions are added. Elements*

*of array\_1 is displayed in line ➃ and array\_2 is displayed in line ➄. The shape of array\_1 is*

*(4, 5) ➅ and array\_2 is (5,) ➆.*

*array\_1.shape → (4, 5)*

*array\_2.shape → (5,)*

*Since array\_2 has less dimension compared to array\_1, according to Rule 1, array\_2 is padded*

*with 1’s on its left. Now the shape of array\_2 becomes (1, 5). NumPy automatically*

*handles this step.*

*array\_1.shape (4, 5)*

*array\_2.shape (1, 5)*

*Next, according to Rule 2, the shape of array\_2 having “1” in the first dimension is stretched*

*to match the highest shape along that dimension of array\_1. The shape of array\_2 becomes*

*(4, 5). NumPy automatically handles this step.*

*array\_1.shape → (4, 5)*

*array\_2.shape → (4, 5)*

*After stretching, the elements of array\_2 seems to be stacked upon themselves for four times*

*along the first dimension. The elements of array\_2 appear to be the copies of the original array.*

*array\_2 → array([****[0, 1, 2, 3, 4]****,*

*[0, 1, 2, 3, 4],*

*[0, 1, 2, 3, 4],*

*[0, 1, 2, 3, 4]])*

*This stretching of the array elements is purely conceptual and does not actually happen as*

*NumPy is smart enough not to make duplicate copies of the original array elements. Also,*

*the original array is not affected.*

*The final shape of array\_2 becomes (4, 5) matching the shape of array\_1, thus paving the*

*way for NumPy to perform an addition operation on these two arrays ➇.*

*Example-2.*

*1. >>> import numpy as np*

*2. >>> array\_1 = np.random.random(4).reshape([4,1])*

*3. >>> array\_2 = np.arange(4)*

*4. >>> array\_1.shape*

*(4, 1)*

*5. >>> array\_2.shape*

*(4,)*

*6. >>> array\_1 + array\_2*

*array([[0.20188425, 1.20188425, 2.20188425, 3.20188425],*

*[0.51342227, 1.51342227, 2.51342227, 3.51342227],*

*[0.03364189, 1.03364189, 2.03364189, 3.03364189],*

*[0.6176858 , 1.6176858 , 2.6176858 , 3.6176858 ]])*

*In the above code, the shape of array\_1 ➁ is (4, 1) ➃ and array\_2 ➂ is (4,) ➄.*

*array\_1.shape → (4, 1)*

*array\_2.shape → (4,)*

*Since array\_2 has less dimension compared to array\_1, according to Rule 1, array\_2 is padded*

*with 1’s on its left. Now the shape of array\_2 becomes (1, 4). NumPy automatically*

*handles this step.*

*array\_1.shape (4, 1)*

*array\_2.shape (1, 4)*

*Next, according to Rule 2, the shape of array\_1 having “1” in the second dimension is*

*stretched to match the highest shape along that dimension of array\_2. Thus, the shape*

*of array\_1 becomes (4, 4). The shape of array\_2 having “1” in the first dimension is*

*stretched to match the highest shape along that dimension of array\_1. Thus, the shape of*

*array\_2 becomes (4, 4). NumPy automatically handles this step.*

*array\_1.shape → (4, 4)*

*array\_2.shape → (4, 4)*