

Introduction to NumPy

Table of Contents

1. What is *NumPy*
2. *NumPy* Array Operations
3. Mathematical Functions
4. Array Manipulation

What is NumPy?

NumPy is a Python library used for working with arrays. It also has functions for working in domain of linear algebra, Fourier transform, and matrices. *NumPy* was created in 2005 by Travis Oliphant. It is an open-source project, and we can use it freely. *NumPy* stands for Numerical Python.

In Python, we have lists that serve the purpose of arrays, but they are slow to process. *NumPy* aims to provide an array object that is up to 50x faster than traditional Python lists. The array object in *NumPy* is called *ndarray*, and it provides a lot of supporting functions that make working with *ndarray* easy.

Creating NumPy Array

A *NumPy* array is similar to a list. It's usually fixed in size and each element is of the same type. We can cast a list to a *NumPy* array by first importing *numpy*:

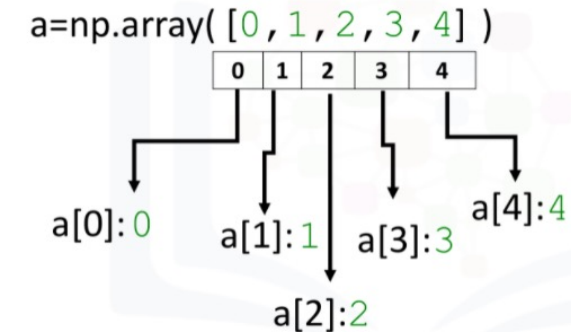
```
Ex: import numpy as np
    a = np.array([0, 1, 2, 3, 4])
    >>>array([0, 1, 2, 3, 4])
```

As with lists, we can access each element via a square bracket:

```
Ex: print("a[0]:", a[0])
    print("a[1]:", a[1])
    print("a[2]:", a[2])

    >>>a[0]: 0
    >>>a[1]: 1
    >>>a[2]: 2
```

Each element is of the same type, in this case integers:



NumPy Array: type, dtype

`type()` method returns class type. If we check the type of the array, we get *numpy.ndarray*:

```
Ex: print(type(a))  
>>>numpy.ndarray
```

As *NumPy* arrays contain data of the same type, we can use the attribute *dtype* to obtain the data type of the array's elements. In this case, it's a 64-bit integer:

```
Ex: print(a.dtype)  
>>>dtype('int64')
```

We can create a *NumPy* array with real numbers. When we check the type of the array, we get *numpy.ndarray*. If we examine the attribute *dtype* we see float 64, as the elements are not integers:

```
Ex: b = np.array([3.1, 11.02, 6.2, 213.2, 5.2])  
    print(type(b))  
    >>>numpy.ndarray  
    print(b.dtype)  
    >>>dtype('float64')
```

NumPy Array: Assign Value

We can change the value of the array. Let's consider the array "c". We can change the first element of the array to 100 as follows:

```
Ex: np.array([20, 1, 2, 3, 4])  
    c[0] = 100  
    print(c)  
    >>>array([100, 1, 2, 3, 4])
```

Similarly, We can change the 5th element of the array to 0 as follows:

```
Ex: c[4] = 0  
    print(c)  
    >>>array([100, 1, 2, 3, 0])
```

NumPy Array: Slicing

Like lists, we can slice the *NumPy* array. We can select the elements from 1 to 3 and assign it to a new *NumPy* array “d” as follows:

```
Ex: c = np.array([20, 1, 2, 3, 4])  
    d = c[1:4]  
    print(d)  
    >>>array([1, 2, 3])
```

We can also assign the corresponding indexes to new values as follows:

```
Ex: c[3:5] = 300,400  
    print(c)  
    >>>array([ 20,  1,  2, 300, 400])
```

NumPy Array: Attributes

Now, let's look at methods for programmatically inspecting an array's attributes (e.g. its size, dimensionality and shape)

```
Ex: a = np.array([0, 1, 2, 3, 4])  
    print(a.size)  
    >>>5
```

The attribute *size* is the number of elements in the array.

The next two attributes will make more sense when we get to higher dimensions but let's review them.

```
Ex: print(a.ndim)  
    >>> 1  
    print(a.shape)  
    >>> (5,)
```

The attribute *ndim* represents the number of array dimensions, or the rank of the array. The attribute *shape* is a tuple of integers indicating the size of the array in each dimension.

NumPy Array: More Attributes

Following are some useful array attributes that are used to describe array and its items:

```
Ex: a = np.array([1, -1, 1, -1])  
    print(a.mean())  
    >>>0.0
```

The *mean()* functions returns the mean value of all the items of the array.

```
Ex: standard_deviation = a.std()  
    print(standard_deviation)  
    >>>1.0
```

std() function returns standard deviation of *NumPy* array.

```
Ex: b = np.array([-1, 2, 3, 4, 5])  
    print(b.max())  
    >>>5
```

Finally, the *max()* and *min()* functions return the biggest and smallest value in the *NumPy* array, respectively.

```
Ex: b = np.array([-1, 2, 3, 4, 5])  
    print(b.min())  
    >>>-1
```

NumPy Array Operations : Addition

In this section, we will see some basic operations that we can perform on *NumPy* arrays. To do so, let's consider the *NumPy* arrays “u” and “v”. We can add the two arrays and assign it to “z”.

```
Ex: u = np.array([1, 0])  
    v = np.array([0, 1])  
    z = u + v  
    print(z)  
    >>>array([1, 1])
```

NumPy Array Operations : Multiplication

Consider the *NumPy* array “y”:

```
Ex: y = np.array([1, 2])  
    z = 2 * y  
    print(z)  
    >>>array([2, 4])
```

We can multiply every element in the array by 2.

Consider the *NumPy* arrays and “v” and “y”:

```
Ex: u = np.array([1, 2])  
    v = np.array([3, 2])  
    z = u * v  
    print(z)  
    >>>array([3, 4])
```

Here we are calculating the production of two *NumPy* arrays.

Similarly, we can calculate dot product:

```
Ex: print(np.dot(u, v))  
    >>>7
```

Converting List to Array

There is a way to convert a list into array. Let's see the following example:

```
Ex: a = [1, 2]  
    np.asarray(a)  
    >>>array([1,2])
```

NumPy: linspace

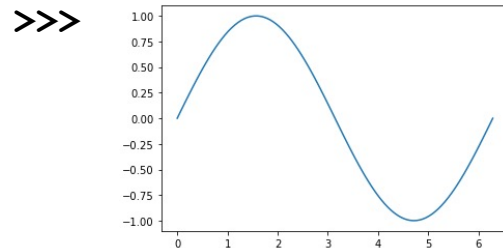
A useful function for plotting mathematical functions is *linspace*. *linspace* returns evenly spaced numbers over a specified interval. We specify the starting point of the sequence and the ending point of the sequence. The parameter *num* indicates the number of samples to generate, in this case 5:

```
Ex: z = np.linspace(-2, 2, num = 5)
    print(z)
    >>>array([-2., -1., 0., 1., 2.] )
```

Making a *NumPy* array within $[-2, 2]$ and 5 elements.

We can use the function *linspace* to generate 100 evenly spaced samples from the interval 0 to 2π :

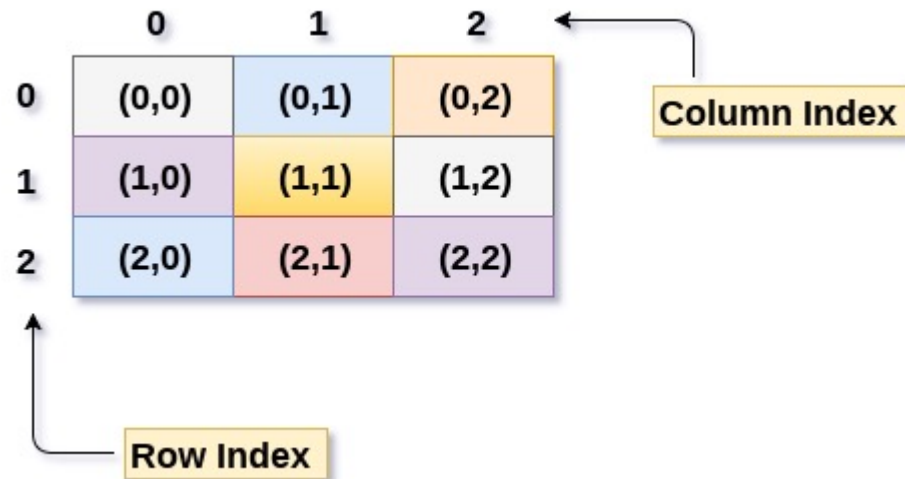
```
Ex: import matplotlib.pyplot as plt
    x = np.linspace(0, 2 * np.pi, num = 100)
    y = np.sin(x)
    plt.plot(x, y)
```



Making a *NumPy* array within $[0, 2\pi]$ and 100 elements and calculating the sine of “x” list. For plotting, we must import the *plt* function from another library of python called *matplotlib*. We will see more detailed use of *Matplotlib*.

2D Arrays in NumPy

2D Array can be defined as array of an array. 2D array are also called as matrices which can be represented as collection of rows and columns.



The diagram illustrates a 3x3 2D array (matrix) with row and column indices. The rows are indexed 0, 1, and 2 from top to bottom. The columns are indexed 0, 1, and 2 from left to right. Each element in the matrix is represented as a tuple (row index, column index). The matrix is color-coded: (0,0) is light blue, (0,1) is light orange, (0,2) is light green, (1,0) is light purple, (1,1) is light yellow, (1,2) is light grey, (2,0) is light blue, (2,1) is light red, and (2,2) is light purple. A yellow box labeled 'Column Index' has an arrow pointing to the column headers. A yellow box labeled 'Row Index' has an arrow pointing to the row headers.

	0	1	2
0	(0,0)	(0,1)	(0,2)
1	(1,0)	(1,1)	(1,2)
2	(2,0)	(2,1)	(2,2)

Creating a 2D NumPy Array

Consider the list `a`, which contains three nested lists each of equal size.

```
Ex: a = [[11, 12, 13], [21, 22, 23], [31, 32, 33]]  
     print(a)  
     >>> [[11, 12, 13], [21, 22, 23], [31, 32, 33]]
```

We can cast the list to a *NumPy* array as follows:

```
Ex: A = np.array(a)  
     print(A)  
     >>> array([[11, 12, 13],  
                [21, 22, 23],  
                [31, 32, 33]])
```

Here, we are converting list to *NumPy* array where every element is the same type.

2D NumPy Array: *ndim*, *shape*, *size*

Like earlier, we can use the attribute *ndim*, *shape* and *size* in 2D arrays.

```
Ex: print(A.ndim)
     print(A.shape)
     print(A.size)
     >>>2
     >>>(3,3)
     >>>9
```

Here, *ndim* returns the number of axes or dimensions, *shape* returns a tuple corresponding to the size or number of each dimension and *size* returns the total number of elements in the array.

Accessing Different Elements

We can use rectangular brackets to access the different elements of the array. The correspondence between the rectangular brackets and the list and the rectangular representation is shown in the following figure for a 3x3 array:

A: [[A[0, 0], A[0, 1], A[0, 2]], [A[1, 0], A[1, 1], A[1, 2]], A[2, 0], A[2, 1], A[2, 2]]]

$$\begin{pmatrix} A[0, 0] & A[0, 1] & A[0, 2] \\ A[1, 0] & A[1, 1] & A[1, 2] \\ A[2, 0] & A[2, 1] & A[2, 2] \end{pmatrix}$$

We can access the 2nd-row, 3rd column as shown in the following figure:

	0	1	2
0	11	12	13
1	21	22	23
2	31	32	33

```
Ex: print(A[1, 2])  
>>>23  
print(A[1][2])  
>>>23
```

2D Array : Slicing

Slicing can be done with the following syntax:

```
Ex: print(A[0][0:2])  
>>>array([11, 12])
```

Here, we are accessing the elements on the first row and first and second columns.

Similarly, we can obtain the first two rows of the 3rd column as follows:

```
Ex: print(A[0:2, 2])  
>>> array([13, 23])
```

Basic Operations: Addition

We can also add arrays. The process is identical to matrix addition. Matrix addition of “X” and “Y” is shown in the following figure:

$$\mathbf{X} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \quad \mathbf{Y} = \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix}$$
$$\mathbf{X} + \mathbf{Y} = \begin{bmatrix} 1+2 & 0+1 \\ 0+1 & 1+2 \end{bmatrix} = \begin{bmatrix} 3 & 1 \\ 1 & 3 \end{bmatrix}$$

We can add *NumPy* arrays as follows:

```
Ex: X = np.array([[1, 0], [0, 1]])
    Y = np.array([[2, 1], [1, 2]])
    Z = X + Y
    print(Z)
    >>>array([[3, 1], [1, 3]])
```

Multiplication 1

Multiplying a *NumPy* array by a scalar is identical to multiplying a matrix by a scalar. For example:

```
Ex: Y = np.array([[2, 1], [1, 2]])  
    Z = 2 * Y  
    print(Z)  
    >>>array([[4, 2], [2, 4]])
```

Multiplication of two arrays corresponds to an element-wise product or Hadamard product.

```
Ex: Y = np.array([[2, 1], [1, 2]])  
    X = np.array([[1, 0], [0, 1]])  
    Z = X * Y  
    print(Z)  
    >>>array([[2, 0], [0, 2]])
```

Here, we are performing element-wise product of the array “X” and “Y”.

Multiplication 2

We can also perform matrix multiplication with the *NumPy* arrays “A” and “B” as follows:

```
Ex: A = np.array([[0, 1, 1], [1, 0, 1]])  
    B = np.array([[1, 1], [1, 1], [-1, 1]])  
    Z = np.dot(A, B)  
    print(Z)  
    >>>array([[0, 2], [0, 2]])
```

Here, we are taking dot product of “X” and “Y”.

We use the *NumPy* attribute *T* to calculate the transposed matrix.

```
Ex: C = np.array([[1, 1], [2, 2], [3, 3]])  
    print(C.T)  
    >>>array([[1, 2, 3], [1, 2, 3]])
```

The transpose of a matrix (2-D array) is simply a flipped version of the original matrix where its rows switch with its columns.

NumPy Array Manipulation: `arange()` , `empty()`

arange() returns evenly spaced values within a given interval:

```
Ex: print(np.arange(3))  
    >>> array([0, 1, 2])  
  
    print(np.arange(3.0))  
    >>> array([ 0.,  1.,  2.])  
  
    print(np.arange(3,7))  
    >>>array([3, 4, 5, 6])
```

empty() returns a new array of given shape and type, without initializing entries.

```
Ex: print(np.empty([2, 2]))  
    >>> array([[4.65455747e-310, 0.00000000e+000],  
               [0.00000000e+000, 0.00000000e+000]])
```

NumPy Array Manipulation: zeros(), ones(), full()

`ones()` returns a new array of given shape and type, filled with ones.

```
Ex: x = np.ones(5)
    print(x)
    >>> array([1., 1., 1., 1., 1.]
```

```
Ex: x = np.ones((2, 1))
    print(x)
    >>> array(array([[1.], [1.]])
```

`zeros()` returns a new array of given shape and type, filled with zeros

```
Ex: x = np.zeros(5)
    print(x)
    >>> array([ 0.,  0.,  0.,  0.,  0.]
```

```
Ex: x = np.zeros((2, 1))
    print(x)
    >>> array([[ 0.] , [0.]])
```

`full()` returns a new array of given shape and type, filled with *fill_value*

```
Ex: x = np.full((2, 2), 1)
    print(x)
    >>> array([[1, 1], [1, 1]])
```

```
Ex: x = np.full((2, 2), [1, 2])
    print(x)
    >>> array([[1, 2], [1, 2]])
```

NumPy Array Manipulation : reshape(), ravel(), transpose()

reshape() gives a new shape to an array without changing its data. Let's see the following example:

```
Ex: a = np.arange(6).reshape((3, 2))  
    print(a)  
    >>> array([[0, 1],  
               [2, 3],  
               [4, 5]])
```

ravel() return a flattened array like to following example:

```
Ex: x = np.array(a)  
    np.ravel(x)  
    >>> array([0, 1, 2, 3, 4, 5])
```

transpose() reverse or permute the axes of an array and returns the modified array.

```
Ex: x = np.arange(4).reshape((2,2))  
    np.transpose(x)  
    >>> array([[0, 2],  
               [1, 3]])
```


NumPy Array Manipulation : repeat()

repeat() repeats elements of an array. Let's see the following example:

```
Ex: r = np.repeat(3, 4)
    >>> array([3, 3, 3, 3])
```

Instead of repeating one element it can repeat multiple elements too

```
Ex: x = np.array([[1,2],[3,4]])
    print(np.repeat(x, 2))
    >>> array([1, 1, 2, 2, 3, 3, 4, 4])
```

We can add another argument called *axis*. Let's see the following example:

```
Ex: x = np.repeat(x, 3, axis=1)
    print(x)
    >>> array([[1, 1, 1, 2, 2, 2],
               [3, 3, 3, 4, 4, 4]])
```

axis : The axis along which to repeat values. By default, use the flattened input array, and return a flat output array. Try to see what changes it shows when *axis* = 0

NumPy Array Manipulation : delete()

Delete() returns a new array with sub-arrays along an axis deleted :

```
Ex: arr = np.array([[1,2,3,4],[5,6,7,8],[9,10,11,12]])  
    np.delete(arr, 1, 0)  
    >>> array([[ 1,  2,  3,  4],[ 9, 10, 11, 12]])
```

Similarly,

```
Ex: arr = np.array([[1,2,3,4],[5,6,7,8],[9,10,11,12]])  
    np.delete(arr, 0, 0)  
    >>> array([[ 5,  6,  7,  8],[ 9, 10, 11, 12]])
```

NumPy Array Manipulation : insert(), append()

insert() inserts values along the given axis before the given indices:

```
Ex: a = np.array([[1, 1], [2, 2], [3, 3]])  
    np.insert(a, 1, 5)  
    >>> array([1, 5, 1, 2, 2, 3, 3])
```

append() appends values at the end of the array:

```
Ex: print(np.append([1, 2, 3], [[4, 5, 6], [7, 8, 9]]))  
    >>> array([1, 2, 3, 4, 5, 6, 7, 8, 9])
```

NumPy Array Manipulation : unique()

unique() finds the unique element of the array:

```
Ex: print( np.unique([1, 1, 2, 2, 3, 3]))  
>>>array([1, 2, 3])
```

Following example return the unique rows of a 2D array. axis = 0 defines row

```
Ex: a = np.array([[1, 0, 0], [1, 0, 0], [2, 3, 4]])  
    print(np.unique(a, axis=0))  
>>> array([[1, 0, 0], [2, 3, 4]])
```

And this one returns the indices of the original array that give the unique values:

```
Ex: a = np.array(['a', 'b', 'b', 'c', 'a'])  
    u, indices = np.unique(a, return_index=True)  
    print(indices)  
>>> array([0, 1, 3])
```

NumPy Array Manipulation : `isin()`

Structure: `numpy.isin(element, test_elements, assume_unique=False, invert=False)`

`isin()` calculates *element* in *test_elements*, broadcasting over *element* only. Returns a boolean array of the same shape as *element* that is True where an element of *element* is in *test_elements* and False otherwise.

```
Ex: element = [[0,2] [4,6]]
    test_elements = [1, 2, 4, 8]
    mask = np.isin(element, test_elements)
    print(mask)
    >>> array([[False,  True],
               [ True, False]])
```