

The Basics of NumPy Arrays

Data manipulation in Python is nearly synonymous with NumPy array manipulation. The following is categories of basic array manipulations:

- i). *Indexing of arrays*: - Getting and setting the value of individual array elements
- ii). *Reshaping of arrays*:- Changing the shape of a given array
- iii). *Slicing of arrays*:- Getting and setting smaller subarrays within a larger array
- iv). *Joining of arrays*:- Combining multiple arrays into one.

Array Indexing: Accessing Single Elements

NumPy indexing is similar to Python's standard list indexing. In a one-dimensional array, you can access the *i*th value (counting from zero) by specifying the desired index in square brackets. You need to *seed* with a set value in order to ensure that the same random arrays are generated each time the code is run.

```
In[1]: import numpy as np
np.random.seed(0) # seed for reproducibility
x1 = np.random.randint(10, size=6) # One-dimensional array
x2 = np.random.randint(10, size=(3, 4)) # Two-dimensional array
x3 = np.random.randint(10, size=(3, 4, 5)) # Three-dimensional
```

```
In[2]: x1
Out[2]: array([5, 0, 3, 3, 7, 9])
```

```
In[3]: x1[0]
Out[3]: 5
```

To index from the end of the array, you need to use negative indices as shown below

```
In[4]: x1[-1]
Out[4]: 9
In[5]: x1[-2]
Out[5]: 7
```

In a multidimensional array, you access items using a comma-separated tuple of indices:

```
In[6]: x2
Out[6]: array([[3, 5, 2, 4],
               [7, 6, 8, 8],
               [1, 6, 7, 7]])
```

```
In[7]: x2[0, 0]
Out[7]: 3
```

```
In[8]: x2[2, 0]
Out[8]: 1
```

```
In[9]: x2[2, -1]
Out[9]: 7
```

You can modify values using any of the above index notation:

```
In[10]: x2[0, 0] = 12
x2
```

```
Out[10]: array([[12, 5, 2, 4],
               [ 7, 6, 8, 8],
               [ 1, 6, 7, 7]])
```

With higher dimensional arrays, you have more options. In a two-dimensional array, the elements at each index are no longer scalars but rather one-dimensional arrays:

```
In[11]: arr2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
In[12]: arr2d[2]
Out[12]: array([7, 8, 9])
```

Reshaping of Arrays

Another useful type of operation is reshaping of arrays. The most flexible way of doing this is with the `reshape()` method. E.g., to put the numbers 1 through 9 in a 3×3 grid as shown below:

```
In[13]: grid = np.arange(1, 10).reshape((3, 3))
        print(grid)
Out[13]:
[[1 2 3]
 [4 5 6]
 [7 8 9]]
```

Note that for this to work, the size of the initial array must match the size of the reshaped array. Another common reshaping pattern is the conversion of a one-dimensional array into a two-dimensional row or column matrix. You can do this with the `reshape` method, or more easily by making use of the *newaxis* keyword within a slice operation:

```
In[14]: x = np.array([1, 2, 3])
# row vector via reshape
x.reshape((1, 3))
Out[14]: array([[1, 2, 3]])

In[15]: # row vector via newaxis
x[np.newaxis, :]
Out[15]: array([[1, 2, 3]])

In[16]: # column vector via reshape
x.reshape((3, 1))
Out[16]: array([[1],
               [2],
               [3]])

In[17]: # column vector via newaxis
x[:, np.newaxis]
Out[17]: array([[1],
               [2],
               [3]])
```

Array Slicing: Accessing Subarrays

Array slicing is used to retrieve subsets of arrays. Just as you can use square brackets to access individual array elements, you can also use them to access subarrays with the *slice* notation, marked by the colon (:) character. The NumPy slicing syntax follows that of the standard Python list; to access a slice of an array `x`, use the following notation: `x[start:stop:step]`

If any of these are unspecified, they default is start=0, stop=*size of dimension*, step=1. Both one dimension and multiple dimension will be considered.

Example

Consider the following one-dimensional array.

```
In[18]: x = np.arange(10)
        x
Out[18]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

In[19]: x[:5] # first five elements
Out[19]: array([0, 1, 2, 3, 4])

In[20]: x[5:] # elements after index 5
Out[21]: array([5, 6, 7, 8, 9])

In[22]: x[4:7] # middle subarray
Out[22]: array([4, 5, 6])

In[23]: x[::2] # every other even element
Out[23]: array([0, 2, 4, 6, 8])

In[24]: x[1::2] # every other odd element, starting at index 1
Out[24]: array([1, 3, 5, 7, 9])
```

When the step value is negative, the defaults for start and stop are swapped. This becomes a convenient way to reverse an array:

```
In[25]: x[::-1] # all elements, reversed
Out[25]: array([9, 8, 7, 6, 5, 4, 3, 2, 1, 0])

In[26]: x[5::-2] # reversed every other from index 5
Out[26]: array([5, 3, 1])
```

If you assign a scalar value to a slice, e.g. `x[5:8] = 12`, the value is propagated (or *broadcasted* henceforth) to the entire selection. Also an important distinction from lists is that array slices are *views* on the original array. This means that the data is not copied, and any modifications to the view will be reflected in the source array:

```
In[27]: x[3:6]=12
        x
Out[27]: array([ 0,  1,  2, 12, 12, 12,  6,  7,  8,  9])
```

Multidimensional subarrays

Multidimensional slices work in the same way, with multiple slices separated by commas.

Examples:

```
In[28]: x2
Out[28]: array([[12,  5,  2,  4],
                [ 7,  6,  8,  8],
                [ 1,  6,  7,  7]])
```

```
In[29]: x2[:2, :3] # two rows, three columns
```

```
Out[29]: array([[12, 5, 2],
               [ 7, 6, 8]])
```

```
In[30]: x2[:3, ::2] # all rows, every other column
```

```
Out[30]: array([[12, 2],
               [ 7, 8],
               [ 1, 7]])
```

```
In[31]: x2[:, ::2] # all columns, every other row
```

```
Out[31]: array([[12, 5, 2, 4],
               [1, 6, 7, 7]])
```

Note that subarray dimensions can be reversed together

```
In[32]: x2[::-1, ::-1]
```

```
Out[32]: array([[ 7, 7, 6, 1],
               [ 8, 8, 6, 7],
               [ 4, 2, 5, 12]])
```

Accessing array rows and columns

One commonly needed routine is accessing single rows or columns of an array. You can do this by combining indexing and slicing, using an empty slice marked by a single colon (:) i.e. with higher dimensional objects, it gives you more options as you can slice one or more axes.

```
In[33]: print(x2[:, 0]) # first column of x2
```

```
Out[33]: [12 7 1]
```

```
In[34]: print(x2[0, :]) # first row of x2
```

```
Out[34]: [12 5 2 4]
```

In the case of row access, the empty slice can be omitted for a more compact syntax

```
In[35]: print(x2[0]) # equivalent to x2[0, :]
```

```
Out[35]: [12 5 2 4]
```

This access becomes interesting in the case of an array having more than two dimensions as shown below.

```
In[36]: arr = np.arange(12).reshape(2,2,3)
        arr
```

```
Out[36]:
```

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

```
In[37]: arr[0]
```

```
Out[37]:
```

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

A slice, therefore, selects a range of elements along an axis. You can pass multiple slices just like you can pass multiple indexes as shown in the following examples of numpy expressions and the resulting arrays:

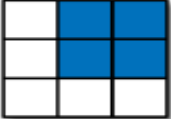
```
In[38]: arr=np.random.randint(10, size=(3,3))
```

```
arr
```

```
Out[38]: array([[7, 2, 9],
               [2, 3, 3],
               [2, 3, 4]])
```

```
In[39]: arr[:2, 1:]
```

```
Out[39]: array([[2, 9],
               [3, 3]])
```


	Expression	Shape
	<code>arr[:2, 1:]</code>	<code>(2, 2)</code>

```
In[40]: arr[2]
```

```
In[40]: arr[2,:]
```

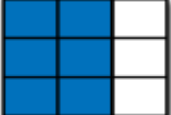
```
In[40]: arr[2:,:]
```

```
Out[40]: array([[2, 3, 4]])
```

	<code>arr[2]</code>	<code>(3,)</code>
	<code>arr[2, :]</code>	<code>(3,)</code>
	<code>arr[2:,:]</code>	<code>(1, 3)</code>

```
In[41]: arr[:, :2]
```

```
Out[41]: array([[7, 2],
               [2, 3],
               [2, 3]])
```


	<code>arr[:, :2]</code>	<code>(3, 2)</code>
---	-------------------------	---------------------

```
In[42]: arr[1,:2]
```

```
Out[42]: array([2, 3])
```

```
In[43]: arr[1:2,:2]
```

```
Out[43]: array([[2, 3]])
```

	<code>arr[1, :2]</code>	<code>(2,)</code>
	<code>arr[1:2, :2]</code>	<code>(1, 2)</code>

If the number of dimensions in the object supplied is less than the dimension of the array being accessed then the colon (:) is assumed for all the dimensions. Consider the following example

```
In [44]: arr = np.arange(12).reshape(2,2,3)
```

```
...: arr
```

```
Out[44]:
array([[[ 0, 1, 2],
        [3, 4, 5]],
```

```
[[6, 7, 8],
 [9, 10, 11]])
```

```
In [45]: arr[1:2]
Out[45]:
array([[ 6,  7,  8],
       [ 9, 10, 11]])
```

Another way to access an array is to use dots (...) based indexing. Suppose in a three- dimensional array you want to access the value of only one column. This can be done in two ways.

```
In [36]: arr = np.arange(27).reshape(3,3,3)
...: arr
Out[46]:
array([[[ 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]]])
```

If we want to access the third column, we can use two different notations to access that column:

```
In [47]: arr[:, :, 2]
Out[47]:
array([[ 2,  5,  8],
       [11, 14, 17],
       [20, 23, 26]])
```

You can also use a dot notation as shown below. Both of the methods gets the same value but the dot notation is concise. The dot notation stands for as many colons as required to complete an indexing operation.

```
In [48]: arr[..., 2]
Out[48]:
array([[ 2,  5,  8],
       [11, 14, 17],
       [20, 23, 26]])
```

Concatenation of arrays

Concatenation, or joining of two arrays in NumPy, is primarily accomplished through the routines `np.concatenate`, `np.vstack`, and `np.hstack`.

- i). `np.concatenate` takes a tuple or list of arrays as its first argument, as shown below:

```
In[49]: x = np.array([1, 2, 3])
        y = np.array([3, 2, 1])
        np.concatenate([x, y])
Out[49]: array([1, 2, 3, 3, 2, 1])
```

You can also concatenate more than two arrays at once:

```
In[50]: z = [99, 99, 99]
```

```
print(np.concatenate([x, y, z]))
Out[50]: [ 1 2 3 3 2 1 99 99 99]
```

np.concatenate can also be used for two-dimensional arrays:

```
In[51]: grid = np.array([[1, 2, 3],
                        [4, 5, 6]])
In[52]: # concatenate along the first axis
np.concatenate([grid, grid])
Out[52]: array([[1, 2, 3],
                [4, 5, 6],
                [1, 2, 3],
                [4, 5, 6]])
In[53]: # concatenate along the second axis
np.concatenate([grid, grid], axis=1)
Out[53]: array([[1, 2, 3, 1, 2, 3],
                [4, 5, 6, 4, 5, 6]])
```

- ii). When working with arrays of mixed dimensions, you will need to use np.vstack (vertical stack) and np.hstack (horizontal stack) functions as shown below:

```
In[54]: x = np.array([1, 2, 3])
        grid = np.array([[9, 8, 7],
                        [6, 5, 4]])

# vertically stack the arrays
In[55]: np.vstack([x, grid])
Out[55]: array([[1, 2, 3],
                [9, 8, 7],
                [6, 5, 4]])
In[56]: # horizontally stack the arrays
y = np.array([[99],
              [99]])
np.hstack([grid, y])
Out[56]: array([[ 9, 8, 7, 99],
                [ 6, 5, 4, 99]])
```

Similarly, np.dstack will stack arrays along the third axis.

Transposing Arrays and Swapping Axes

Transposing is a special form of reshaping which similarly returns a view on the underlying data without copying anything. Arrays have the transpose method and also the special T attribute:

```
In[57]: arr = np.arange(15).reshape((3, 5))
        Arr

Out[57]:
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14]])
In[58]: arr.T
Out[58]:
array([[ 0,  5, 10],
```

```
[ 1,  6, 11],  
[ 2,  7, 12],  
[ 3,  8, 13],  
[ 4,  9, 14]])
```

Random Number Generation

The `numpy.random` module supplements the built-in Python random with functions for efficiently generating whole arrays of sample values from many kinds of probability distributions. For example, you can get a 4 by 4 array of samples from the standard normal distribution using `normal`:

```
In[59]: samples = np.random.normal(size=(4, 4))  
        samples
```

```
Out[59]:  
array([[ -0.560404,  -1.26957996,   0.98812,   1.95250687],  
       [  0.33060678,   0.02592661,  -0.10262176,  -1.31960601],  
       [-0.15359825,  -0.87913398,   0.97153129,   0.61420498],  
       [-0.87504239,   1.02479073,  -2.14685522,  -0.04027206]])
```

The following is a partial list of *numpy.random* functions

Function	Description
Seed	Seed the random number generator
Rand	Draw samples from a uniform distribution
randint	Draw random integers from a given low-to-high range
randn	Draw samples from a normal distribution with mean 0 and standard deviation 1
binomial	Draw samples a binomial distribution
normal	Draw samples from a normal (Gaussian) distribution
beta	Draw samples from a beta distribution
chisquare	Draw samples from a chi-square distribution
gamma	Draw samples from a gamma distribution
uniform	Draw samples from a uniform [0, 1) distribution