

```
In[12]: x2[2, 0]
```

```
Out[12]: 1
```

```
In[13]: x2[2, -1]
```

```
Out[13]: 7
```

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

```
In[14]: x2[0, 0] = 12
        x2
```

```
Out[14]: array([[12,  5,  2,  4],
                [ 7,  6,  8,  8],
                [ 1,  6,  7,  7]])
```

Keep in mind that, unlike Python lists, NumPy arrays have a fixed type. This means, for example, that if you attempt to insert a floating-point value to an integer array, the value will be silently truncated. Don't be caught unaware by this behavior!

```
In[15]: x1[0] = 3.14159 # this will be truncated!
        x1
```

```
Out[15]: array([3, 0, 3, 3, 7, 9])
```

Array Slicing: Accessing Subarrays

Just as we can use square brackets to access individual array elements, we 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 this:

```
x[start:stop:step]
```

If any of these are unspecified, they default to the values `start=0`, `stop=size of dimension`, `step=1`. We'll take a look at accessing subarrays in one dimension and in multiple dimensions.

One-dimensional subarrays

```
In[16]: x = np.arange(10)
        x
```

```
Out[16]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

```
In[17]: x[:5] # first five elements
```

```
Out[17]: array([0, 1, 2, 3, 4])
```

```
In[18]: x[5:] # elements after index 5
```

```
Out[18]: array([5, 6, 7, 8, 9])
```

```
In[19]: x[4:7] # middle subarray
```

```
Out[19]: array([4, 5, 6])
```

```
In[20]: x[::2] # every other element
Out[20]: array([0, 2, 4, 6, 8])

In[21]: x[1::2] # every other element, starting at index 1
Out[21]: array([1, 3, 5, 7, 9])
```

A potentially confusing case is when the step value is negative. In this case, the defaults for `start` and `stop` are swapped. This becomes a convenient way to reverse an array:

```
In[22]: x[::-1] # all elements, reversed
Out[22]: array([9, 8, 7, 6, 5, 4, 3, 2, 1, 0])

In[23]: x[5::-2] # reversed every other from index 5
Out[23]: array([5, 3, 1])
```

Multidimensional subarrays

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

```
In[24]: x2
Out[24]: array([[12,  5,  2,  4],
               [ 7,  6,  8,  8],
               [ 1,  6,  7,  7]])

In[25]: x2[:2, :3] # two rows, three columns
Out[25]: array([[12,  5,  2],
               [ 7,  6,  8]])

In[26]: x2[:3, ::2] # all rows, every other column
Out[26]: array([[12,  2],
               [ 7,  8],
               [ 1,  7]])
```

Finally, subarray dimensions can even be reversed together:

```
In[27]: x2[::-1, ::-1]
Out[27]: 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 (`:`):

```
In[28]: print(x2[:, 0]) # first column of x2
[12  7  1]
```

```
In[29]: print(x2[0, :]) # first row of x2
[12  5  2  4]
```

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

```
In[30]: print(x2[0]) # equivalent to x2[0, :]
[12  5  2  4]
```

Subarrays as no-copy views

One important—and extremely useful—thing to know about array slices is that they return *views* rather than *copies* of the array data. This is one area in which NumPy array slicing differs from Python list slicing: in lists, slices will be copies. Consider our two-dimensional array from before:

```
In[31]: print(x2)
[[12  5  2  4]
 [ 7  6  8  8]
 [ 1  6  7  7]]
```

Let's extract a 2×2 subarray from this:

```
In[32]: x2_sub = x2[:2, :2]
        print(x2_sub)
[[12  5]
 [ 7  6]]
```

Now if we modify this subarray, we'll see that the original array is changed! Observe:

```
In[33]: x2_sub[0, 0] = 99
        print(x2_sub)
[[99  5]
 [ 7  6]]

In[34]: print(x2)
[[99  5  2  4]
 [ 7  6  8  8]
 [ 1  6  7  7]]
```

This default behavior is actually quite useful: it means that when we work with large datasets, we can access and process pieces of these datasets without the need to copy the underlying data buffer.

Creating copies of arrays

Despite the nice features of array views, it is sometimes useful to instead explicitly copy the data within an array or a subarray. This can be most easily done with the `copy()` method:

```
In[35]: x2_sub_copy = x2[:, 2, :2].copy()
        print(x2_sub_copy)
```

```
[[99  5]
 [ 7  6]]
```

If we now modify this subarray, the original array is not touched:

```
In[36]: x2_sub_copy[0, 0] = 42
        print(x2_sub_copy)
```

```
[[42  5]
 [ 7  6]]
```

```
In[37]: print(x2)
```

```
[[99  5  2  4]
 [ 7  6  8  8]
 [ 1  6  7  7]]
```

Reshaping of Arrays

Another useful type of operation is reshaping of arrays. The most flexible way of doing this is with the `reshape()` method. For example, if you want to put the numbers 1 through 9 in a 3×3 grid, you can do the following:

```
In[38]: grid = np.arange(1, 10).reshape((3, 3))
        print(grid)
```

```
[[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. Where possible, the `reshape` method will use a no-copy view of the initial array, but with noncontiguous memory buffers this is not always the case.

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[39]: x = np.array([1, 2, 3])

        # row vector via reshape
        x.reshape((1, 3))
```

```
Out[39]: array([[1, 2, 3]])
```

```
In[40]: # row vector via newaxis
        x[np.newaxis, :]
```

```
Out[40]: array([[1, 2, 3]])
```