NumPy Basics: Arrays and Vectorized Computation

Part 2

The NumPy ndarray: A Multidimensional Array Object

Part 2

Data Types for ndarrays

- The *data type* or `dtype` is a special object containing the information (or metadata, data about data) the ndarray needs to interpret a chunk of memory as a particular type of data:

```
In [22]: arr1 = np.array([1, 2, 3], dtype=np.float64)

In [23]: arr1.dtype
Out[23]: dtype('float64')

In [24]: arr2 = np.array([1, 2, 3], dtype=np.int32)

In [25]: arr2.dtype
Out[25]: dtype('int32')
```

Type	Type code	Description
`int8, uint8`	`i1, u1`	Signed and unsigned 8-bit (1 byte) integer types
`int16, uint16`	`i2, u2`	Signed and unsigned 16-bit integer types
`int32, uint32`	`i4, u4`	Signed and unsigned 32-bit integer types
`int64, uint64`	`i8, u8`	Signed and unsigned 64-bit integer types

`float16`	`f2`	Half-precision floating point
`float32`	`f4` or `f`	Standard single-precision floating point; compatible with C float
`float64`	`f8` or `d`	Standard double-precision floating point; compatible with C double and Python `float` object
`float128`	`f16` or `g`	Extended-precision floating point

`complex64`, `complex128`, `complex256`	`c8`, `c16`, `c32`	Complex numbers represented by two 32, 64, or 128 floats, respectively
`bool`	?	Boolean type storing `True` and `False` values
`object`	O	Python object type; a value can be any Python object
`string_`	S	Fixed-length ASCII string type (1 byte per character); for example, to create a string dtype with length 10, use `'S10'`
`unicode_`	U	Fixed-length Unicode type (number of bytes platform specific); same specification semantics as `string_` (e.g., `'U10'`)

- You can explicitly convert or *cast* an array from one dtype to another using ndarray's `astype` method:

```
In [26]: arr = np.array([1, 2, 3, 4, 5])
         arr.dtype

Out[26]: dtype('int64')

In [27]: float_arr = arr.astype(np.float64)
         float_arr.dtype

Out[27]: dtype('float64')
```

- If I cast some floating-point numbers to be of integer dtype, the decimal part will be truncated:

```
In [28]: arr = np.array([3.7, -1.2, -2.6, 0.5, 12.9, 10.1])
         arr
Out[28]: array([ 3.7, -1.2, -2.6,  0.5, 12.9, 10.1])

In [29]: arr.astype(np.int32)
Out[29]: array([ 3, -1, -2,  0, 12, 10], dtype=int32)
```

- If you have an array of strings representing numbers, you can use `astype` to convert them to numeric form:

```
In [30]: numeric_strings = np.array(['1.25', '-9.6', '42'], dtype=np.string_)
         numeric_strings.astype(float)

Out[30]: array([ 1.25, -9.6 , 42.  ])
```

- You can also use another array's `dtype` attribute:

```
In [31]: int_array = np.arange(10)
         calibers = np.array([.22, .270, .357, .380, .44, .50], dtype=np.float64)
         int_array.astype(calibers.dtype)

Out[31]: array([0., 1., 2., 3., 4., 5., 6., 7., 8., 9.])
```

- There are shorthand type code strings you can also use to refer to a dtype:

```
In [32]: empty_uint32 = np.empty(8, dtype='u4')
         empty_uint32

Out[32]: array([         0, 1075314688,          0, 1075707904,          0,
                1075838976,          0, 1072693248], dtype=uint32)
```

- Calling `astype` always creates a new array (a copy of the data), even if the new dtype is the same as the old dtype.

Arithmetic with NumPy Arrays

- Arrays are important because they enable you to express batch operations on data without writing any for loops.
- NumPy users call this *vectorization*.
- Any arithmetic operations between equal-size arrays applies the operation element-wise:

```
In [33]: arr = np.array([[1., 2., 3.], [4., 5., 6.]])
         arr

Out[33]: array([[1., 2., 3.],
                [4., 5., 6.]])

In [34]: arr * arr

Out[34]: array([[ 1.,  4.,  9.],
                [16., 25., 36.]])

In [35]: arr - arr

Out[35]: array([[0., 0., 0.],
                [0., 0., 0.]])
```

- Arithmetic operations with scalars propagate the scalar argument to each element in the array:

```
In [36]: arr

Out[36]: array([[1., 2., 3.],
                [4., 5., 6.]])

In [37]: 1 / arr

Out[37]: array([[1.        , 0.5       , 0.33333333],
                [0.25      , 0.2       , 0.16666667]])

In [38]: arr ** 0.5

Out[38]: array([[1.        , 1.41421356, 1.73205081],
                [2.        , 2.23606798, 2.44948974]])
```

- Comparisons between arrays of the same size yield boolean arrays:

```
In [39]: arr2 = np.array([[0., 4., 1.], [7., 2., 12.]])
         arr2

Out[39]: array([[ 0.,  4.,  1.],
                [ 7.,  2., 12.]])

In [40]: arr

Out[40]: array([[1., 2., 3.],
                [4., 5., 6.]])

In [41]: arr2 > arr

Out[41]: array([[False,  True, False],
                [ True, False,  True]])
```

Basic Indexing and Slicing

- NumPy array indexing is a rich topic, as there are many ways you may want to select a subset of your data or individual elements.
- One-dimensional arrays are simple; on the surface they act similarly to Python lists:

```
In [42]: arr = np.arange(10)
         arr
Out[42]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

In [43]: arr[5]
Out[43]: 5

In [44]: arr[5:8]
Out[44]: array([5, 6, 7])

In [45]: arr[5:8] = 12
         arr
Out[45]: array([ 0,  1,  2,  3,  4, 12, 12, 12,  8,  9])
```

- As you can see, if you assign a scalar value to a slice, as in `arr[5:8] = 12`, the value is propagated (or *broadcasted* henceforth) to the entire selection.
- An important first distinction from Python's built-in 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.

- To give an example of this, I first create a slice of `arr`:

```
In [46]: arr_slice = arr[5:8]
         arr_slice

Out[46]: array([12, 12, 12])
```

- Now, when I change values in `arr_slice`, the mutations are reflected in the original array `arr`:

```
In [47]: arr_slice[1] = 12345
         arr

Out[47]: array([    0,     1,     2,     3,     4,    12, 12345,    12,     8,
                    9])
```

- The “bare” slice `[:]` will assign to all values in an array:

```
In [48]: arr_slice[:] = 64
         arr

Out[48]: array([ 0,  1,  2,  3,  4, 64, 64, 64,  8,  9])
```

- If you are new to NumPy, you might be surprised by this, especially if you have used other array programming languages that copy data more eagerly.
- As NumPy has been designed to be able to work with very large arrays, you could imagine performance and memory problems if NumPy insisted on always copying data.

- If you want a copy of a slice of an ndarray instead of a view, you will need to explicitly copy the array—for example, `arr[5:8].copy()`.

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

```
In [49]: arr2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
         arr2d[2]

Out[49]: array([7, 8, 9])
```

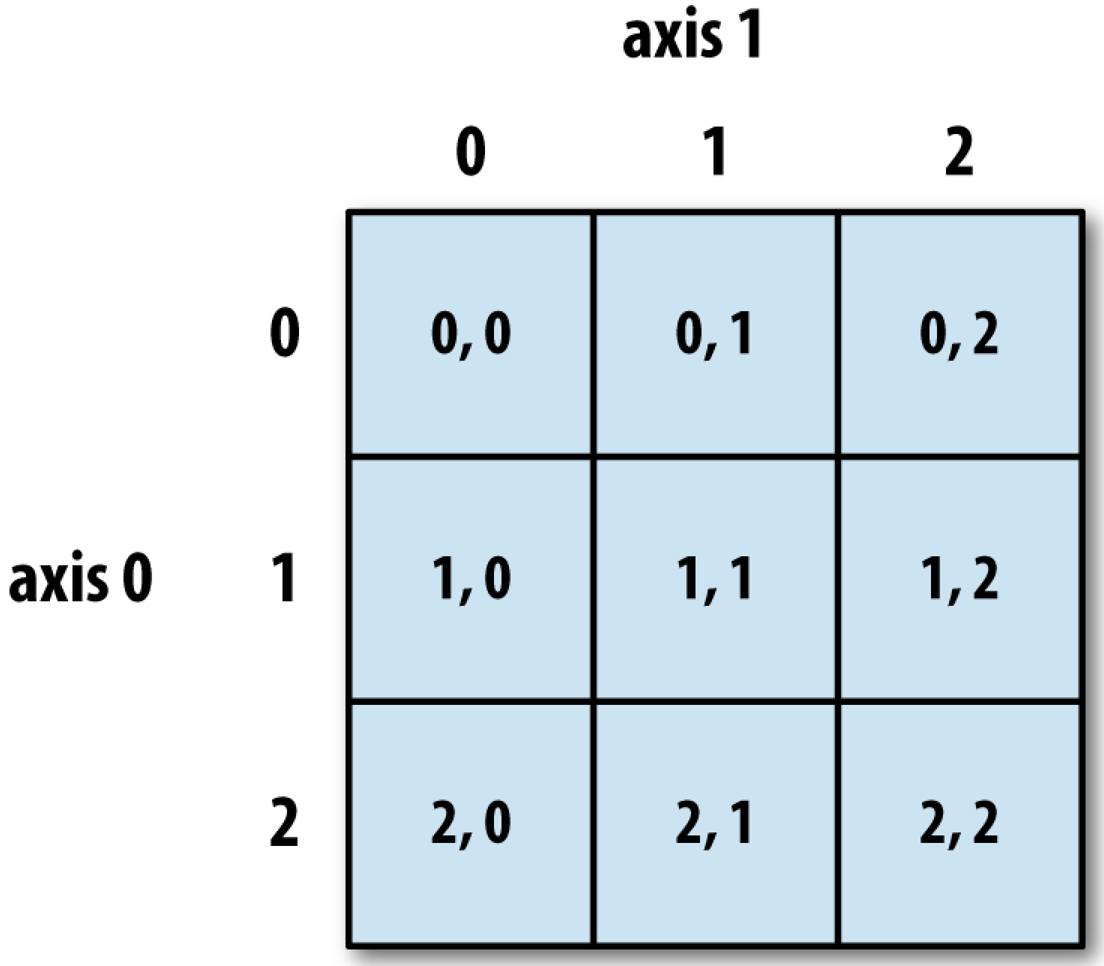
- Thus, individual elements can be accessed recursively.
- But that is a bit too much work, so you can pass a comma-separated list of indices to select individual elements.
- So these are equivalent:

```
In [50]: arr2d
Out[50]: array([[1, 2, 3],
                [4, 5, 6],
                [7, 8, 9]])

In [51]: arr2d[0][2]
Out[51]: 3

In [52]: arr2d[0, 2]
Out[52]: 3
```

- See the following figure for an illustration of indexing on a two-dimensional array.
- I find it helpful to think of axis 0 as the “rows” of the array and axis 1 as the “columns.”



- In multidimensional arrays, if you omit later indices, the returned object will be a lower dimensional ndarray consisting of all the data along the higher dimensions.
- So in the 2 × 2 × 3 array `arr3d`:

```
In [53]: arr3d = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]])
         arr3d

Out[53]: array([[[ 1,  2,  3],
                 [ 4,  5,  6]],

                [[ 7,  8,  9],
                 [10, 11, 12]]])
```

- `arr3d[0]` is a 2 × 3 array:

```
In [54]: arr3d[0]

Out[54]: array([[1, 2, 3],
                [4, 5, 6]])
```

- Both scalar values and arrays can be assigned to `arr3d[0]`:

```
In [55]: old_values = arr3d[0].copy()
         arr3d[0] = 42
         arr3d

Out[55]: array([[[42, 42, 42],
                 [42, 42, 42]],

                [[ 7,  8,  9],
                 [10, 11, 12]]])

In [56]: arr3d[0] = old_values
         arr3d

Out[56]: array([[[ 1,  2,  3],
                 [ 4,  5,  6]],

                [[ 7,  8,  9],
                 [10, 11, 12]]])
```

- Similarly, `arr3d[1, 0]` gives you all of the values whose indices start with `(1, 0)`, forming a 1-dimensional array:

```
In [57]: arr3d
Out[57]: array([[[ 1,  2,  3],
                 [ 4,  5,  6]],

                [[ 7,  8,  9],
                 [10, 11, 12]]])

In [58]: arr3d[1, 0]
Out[58]: array([7, 8, 9])
```

- This expression is the same as though we had indexed in two steps:

```
In [59]: x = arr3d[1]
         x
Out[59]: array([[ 7,  8,  9],
                [10, 11, 12]])

In [60]: x[0]
Out[60]: array([7, 8, 9])
```

- Note that in all of these cases where subsections of the array have been selected, the returned arrays are views.