The Broadcasting Rule

Two arrays are compatible for broadcasting if for each trailing dimension (i.e., starting from the end) the axis lengths match or if either of the lengths is 1. Broadcasting is then performed over the missing or length 1 dimensions.

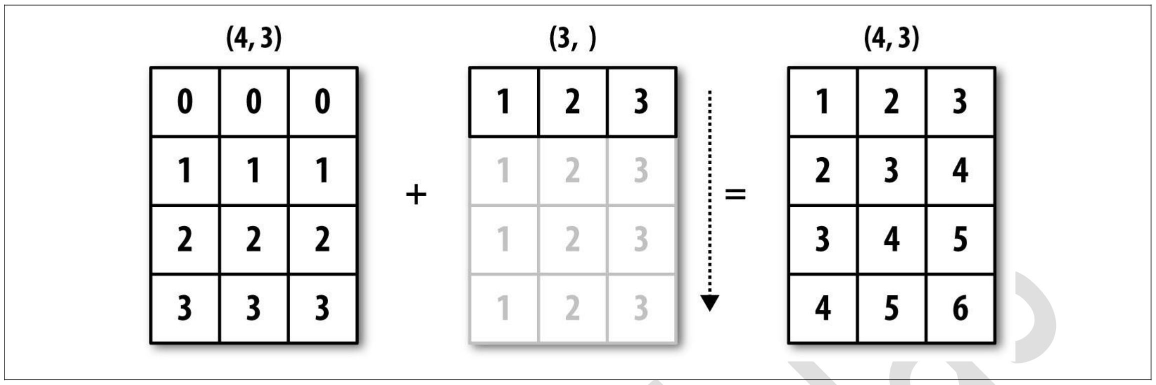

Figure A-4. Broadcasting over axis 0 with a 1D array

Even as an experienced NumPy user, I often find myself having to pause and draw a diagram as I think about the broadcasting rule. Consider the last example and suppose we wished instead to subtract the mean value from each row. Since arr.mean(0) has length 3, it is compatible for broadcasting across axis 0 because the trailing dimension in arr is 3 and therefore matches. According to the rules, to subtract over axis 1 (i.e., subtract the row mean from each row), the smaller array must have shape (4, 1):

```
In [87]: arr Out[87]:
array([[ 0.0009,  1.3438, -0.7135],        [-0.8312, -
2.3702, -1.8608],
       [-0.8608,  0.5601, -1.2659],
       [ 0.1198, -1.0635,  0.3329]])

In [88]: row_means = arr.mean(1)

In [89]: row_means.shape
Out[89]: (4,)

In [90]: row_means.reshape((4, 1))
Out[90]:
array([[ 0.2104],        [-1.6874],
       [-0.5222],
       [-0.2036]])
In [91]: demeaned = arr - row_means.reshape((4, 1))

In [92]: demeaned.mean(1)
Out[92]: array([ 0., -0.,  0.,  0.])
```

See Figure A-5 for an illustration of this operation.

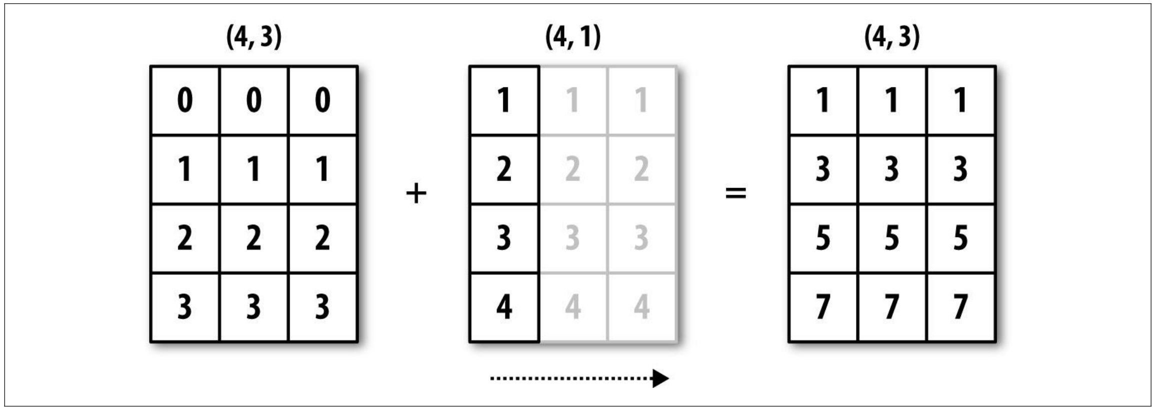

Figure A-5. Broadcasting over axis 1 of a 2D array

See Figure A-6 for another illustration, this time adding a two-dimensional array to a three-dimensional one across axis 0.

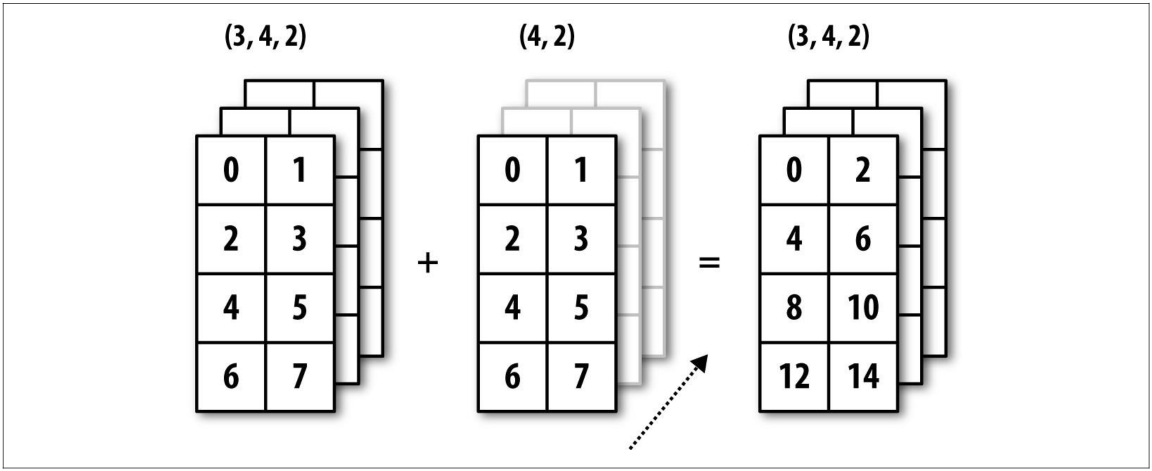

Figure A-6. Broadcasting over axis 0 of a 3D array

Broadcasting Over Other Axes

Broadcasting with higher dimensional arrays can seem even more mind-bending, but it is really a matter of following the rules. If you don't, you'll get an error like this:

```
In [93]: arr - arr.mean(1)
---------------------------------------------------------------------------ValueError
Traceback (most recent call last)
```

```
<ipython-input-93-7b87b85a20b2> in <module>()
----> 1 arr - arr.mean(1)
```

ValueError: operands could not be broadcast together with shapes (4,3) (4,) It's quite common to want to perform an arithmetic operation with a lower dimensional array across axes other than axis 0. According to the broadcasting rule, the "broadcast dimensions" must be 1 in the smaller array. In the example of row demeaning shown here, this meant reshaping the row means to be shape (4, 1) instead of (4,):

```
In [94]: arr - arr.mean(1).reshape((4, 1))
Out[94]:
array([[-0.2095,  1.1334, -0.9239],        [
0.8562, -0.6828, -0.1734],
       [-0.3386,  1.0823, -0.7438],
       [ 0.3234, -0.8599,  0.5365]])
```

In the three-dimensional case, broadcasting over any of the three dimensions is only a matter of reshaping the data to be shape-compatible. Figure A-7 nicely visualizes the shapes required to broadcast over each axis of a three-dimensional array.

A common problem, therefore, is needing to add a new axis with length 1 specifically for broadcasting purposes. Using reshape is one option, but inserting an axis requires constructing a tuple indicating the new shape. This can often be a tedious exercise. Thus, NumPy arrays offer a special syntax for inserting new axes by indexing. We use the special np.newaxis attribute along with "full" slices to insert the new axis:

```
In [95]: arr = np.zeros((4, 4))

In [96]: arr_3d = arr[:, np.newaxis, :]

In [97]: arr_3d.shape
Out[97]: (4, 1, 4)

In [98]: arr_1d = np.random.normal(size=3)

In [99]: arr_1d[:, np.newaxis]
Out[99]: array([[-2.3594],        [-
0.1995],
       [-1.542 ]])

In [100]: arr_1d[np.newaxis, :]
Out[100]: array([[-2.3594, -0.1995, -1.542 ]])
```

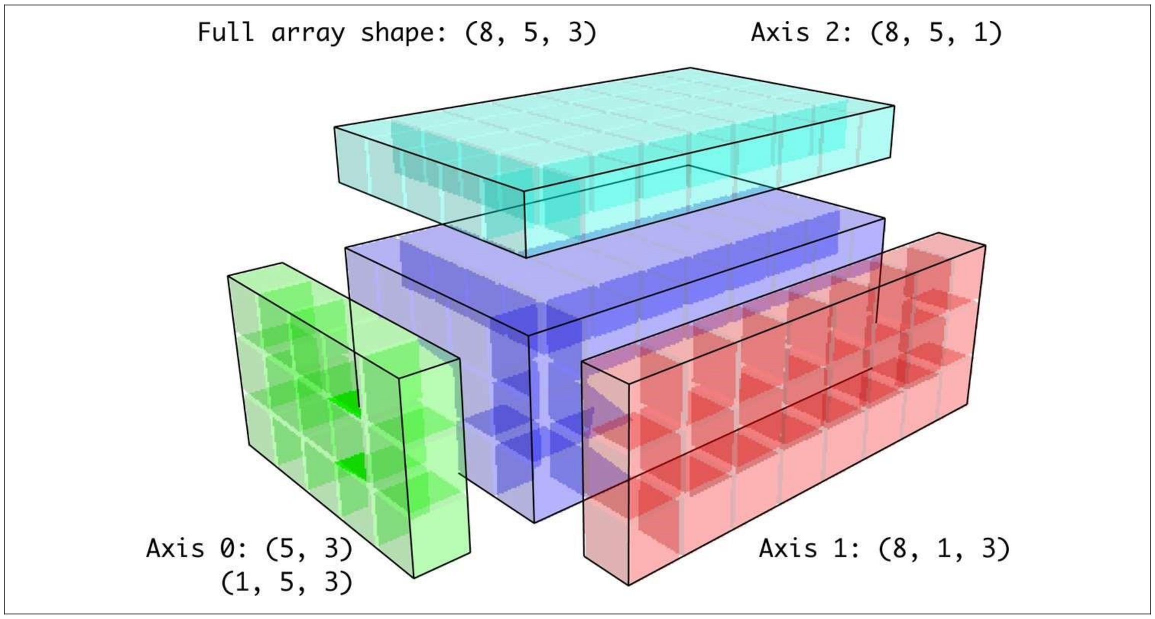

Figure A-7. Compatible 2D array shapes for broadcasting over a 3D array

Thus, if we had a three-dimensional array and wanted to demean axis 2, say, we would need to write:

```
In [101]: arr = np.random.randn(3, 4, 5)

In [102]: depth_means = arr.mean(2)

In [103]: depth_means
Out[103]:
array([[-0.4735,  0.3971, -0.0228,  0.2001],        [-0.3521, -0.281 , -
0.071 , -0.1586],
       [ 0.6245,  0.6047,  0.4396, -0.2846]])

In [104]: depth_means.shape
Out[104]: (3, 4)

In [105]: demeaned = arr - depth_means[:, :, np.newaxis]

In [106]: demeaned.mean(2)
Out[106]: array([[ 0.,   0., -0., -0.],
[ 0.,  0., -0.,  0.],
       [ 0.,  0., -0., -0.]])
```

You might be wondering if there’s a way to generalize demeaning over an axis without sacrificing performance. There is, but it requires some indexing gymnastics:

```
def demean_axis(arr, axis=0):     means =
arr.mean(axis)

    # This generalizes things like [:, :, np.newaxis] to N dimensions     indexer =
[slice(None)] * arr.ndim     indexer[axis] = np.newaxis     return arr - means[indexer]
```

Setting Array Values by Broadcasting

The same broadcasting rule governing arithmetic operations also applies to setting values via array indexing. In a simple case, we can do things like:

```
In [107]: arr = np.zeros((4, 3))

In [108]: arr[:] = 5

In [109]: arr Out[109]:
array([[ 5.,  5.,  5.],       [ 5.,  5.,
5.],
       [ 5.,  5.,  5.],
       [ 5.,  5.,  5.]])
```

However, if we had a one-dimensional array of values we wanted to set into the columns of the array, we can do that as long as the shape is compatible:

```
In [110]: col = np.array([1.28, -0.42, 0.44, 1.6])

In [111]: arr[:] = col[:, np.newaxis]

In [112]: arr
Out[112]:
array([[ 1.28,  1.28,  1.28],       [-0.42, -0.42, -
0.42],
       [ 0.44,  0.44,  0.44],
       [ 1.6 ,  1.6 ,  1.6 ]])

In [113]: arr[:2] = [[-1.37], [0.509]]

In [114]: arr
Out[114]:
array([[-1.37 , -1.37 , -1.37 ],       [ 0.509,
0.509,  0.509],       [ 0.44 ,  0.44 ,  0.44 ],
       [ 1.6  ,  1.6  ,  1.6  ]])
```