Broadcasting

See also:
numpy.broadcast (../reference/generated/numpy.broadcast.html#numpy.broadcast)

The term broadcasting describes how numpy treats arrays with different shapes during arithmetic operations. Subject to certain constraints, the smaller array is "broadcast" across the larger array so that they have compatible shapes. Broadcasting provides a means of vectorizing array operations so that looping occurs in C instead of Python. It does this without making needless copies of data and usually leads to efficient algorithm implementations. There are, however, cases where broadcasting is a bad idea because it leads to inefficient use of memory that slows computation.

NumPy operations are usually done on pairs of arrays on an element-by-element basis. In the simplest case, the two arrays must have exactly the same shape, as in the following example:

```
>>> a = np.array([1.0, 2.0, 3.0])
>>> b = np.array([2.0, 2.0, 2.0])
>>> a * b
array([ 2.,  4.,  6.])
```

NumPy's broadcasting rule relaxes this constraint when the arrays' shapes meet certain constraints. The simplest broadcasting example occurs when an array and a scalar value are combined in an operation:

```
>>> a = np.array([1.0, 2.0, 3.0])
>>> b = 2.0
>>> a * b
array([ 2.,  4.,  6.])
```

The result is equivalent to the previous example where `b` was an array. We can think of the scalar `b` being *stretched* during the arithmetic operation into an array with the same shape as `a`. The new elements in `b` are simply copies of the original scalar. The stretching analogy is only conceptual. NumPy is smart enough to use the original scalar value without actually making copies, so that broadcasting operations are as memory and computationally efficient as possible.

The code in the second example is more efficient than that in the first because broadcasting moves less memory around during the multiplication (`b` is a scalar rather than an array).

General Broadcasting Rules

When operating on two arrays, NumPy compares their shapes element-wise. It starts with the trailing dimensions, and works its way forward. Two dimensions are compatible when

1. they are equal, or
2. one of them is 1

If these conditions are not met, a `ValueError: frames are not aligned` exception is thrown, indicating that the arrays have incompatible shapes. The size of the resulting array is the maximum size along each dimension of the input arrays.

Arrays do not need to have the same *number* of dimensions. For example, if you have a `256x256x3` array of RGB values, and you want to scale each color in the image by a different value, you can multiply the image by a one-dimensional array with 3 values. Lining up the sizes of the trailing axes of these arrays according to the broadcast rules, shows that they are compatible:

```
Image  (3d array): 256 x 256 x 3
Scale  (1d array):             3
Result (3d array): 256 x 256 x 3
```

When either of the dimensions compared is one, the other is used. In other words, dimensions with size 1 are stretched or “copied” to match the other.

In the following example, both the `A` and `B` arrays have axes with length one that are expanded to a larger size during the broadcast operation:

```
A      (4d array):  8 x 1 x 6 x 1
B      (3d array):      7 x 1 x 5
Result (4d array):  8 x 7 x 6 x 5
```

Here are some more examples:

```
A      (2d array):  5 x 4
B      (1d array):      1
Result (2d array):  5 x 4

A      (2d array):  5 x 4
B      (1d array):      4
Result (2d array):  5 x 4

A      (3d array):  15 x 3 x 5
B      (3d array):  15 x 1 x 5
Result (3d array):  15 x 3 x 5

A      (3d array):  15 x 3 x 5
B      (2d array):       3 x 5
Result (3d array):  15 x 3 x 5

A      (3d array):  15 x 3 x 5
B      (2d array):       3 x 1
Result (3d array):  15 x 3 x 5
```

Here are examples of shapes that do not broadcast:

```
A      (1d array):  3
B      (1d array):  4 # trailing dimensions do not match

A      (2d array):      2 x 1
B      (3d array):  8 x 4 x 3 # second from last dimensions mismatched
```

An example of broadcasting in practice:

```
>>>
>>> x = np.arange(4)
>>> xx = x.reshape(4,1)
>>> y = np.ones(5)
>>> z = np.ones((3,4))

>>> x.shape
(4,)

>>> y.shape
(5,)

>>> x + y
<type 'exceptions.ValueError'>: shape mismatch: objects cannot be broadcast to a single sha
pe

>>> xx.shape
(4, 1)

>>> y.shape
(5,)

>>> (xx + y).shape
(4, 5)

>>> xx + y
array([[ 1.,  1.,  1.,  1.,  1.],
       [ 2.,  2.,  2.,  2.,  2.],
       [ 3.,  3.,  3.,  3.,  3.],
       [ 4.,  4.,  4.,  4.,  4.]])

>>> x.shape
(4,)

>>> z.shape
(3, 4)

>>> (x + z).shape
(3, 4)

>>> x + z
array([[ 1.,  2.,  3.,  4.],
       [ 1.,  2.,  3.,  4.],
       [ 1.,  2.,  3.,  4.]])
```

Broadcasting provides a convenient way of taking the outer product (or any other outer operation) of two arrays. The following example shows an outer addition operation of two 1-d arrays:

```
>>> a = np.array([0.0,  10.0,  20.0,  30.0])
>>> b = np.array([1.0,  2.0,  3.0])
>>> a[:,  np.newaxis] + b
array([[  1.,    2.,    3.],
       [ 11.,   12.,   13.],
       [ 21.,   22.,   23.],
       [ 31.,   32.,   33.]])
```

Here the `newaxis` index operator inserts a new axis into `a`, making it a two-dimensional `4x1` array. Combining the `4x1` array with `b`, which has shape `(3,)`, yields a `4x3` array.

See this article (http://wiki.scipy.org/EricsBroadcastingDoc) for illustrations of broadcasting concepts.

Previous topic

Indexing (basics.indexing.html)

Next topic

Byte-swapping (basics.byteswapping.html)