Computation on Arrays: Broadcasting

在数组上计算：广播

> We saw in the previous section how NumPy's universal functions can be used to *vectorize* operations and thereby remove slow Python loops. Another means of vectorizing operations is to use NumPy's *broadcasting* functionality. Broadcasting is simply a set of rules for applying binary ufuncs (e.g., addition, subtraction, multiplication, etc.) on arrays of different sizes.

我们在前面的章节中学习了NumPy的通用函数，它们用来对数组进行*向量化*操作，从而抛弃了性能低下的Python循环。还有一种对NumPy数组进行向量化操作的方式我们称为*广播*。广播简单来说就是一整套用于在不同尺寸或形状的数组之间进行二元ufuncs运算（如加法、减法、乘法等）的规则。

Introducing Broadcasting

广播简介

> Recall that for arrays of the same size, binary operations are performed on an element-by-element basis:

回忆一下对于相同尺寸的数组来说，二元运算是按每个元素进行运算的：

In [1]:
```
import numpy as np
```

In [2]:
```
a = np.array([0, 1, 2])
b = np.array([5, 5, 5])
a + b
```

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

> Broadcasting allows these types of binary operations to be performed on arrays of different sizes–for example, we can just as easily add a scalar (think of it as a zero-dimensional array) to an array:

广播机制允许这样的二元运算能够在不同尺寸和形状的数组之间进行，例如，我们可以用数组和一个标量相加（标量可以认为是一个零维数组）：

In [3]:
```
a + 5
```

Out[3]:
```
array([5, 6, 7])
```

> We can think of this as an operation that stretches or duplicates the value `5` into the array `[5, 5, 5]`, and adds the results. The advantage of NumPy's broadcasting is that this duplication of values does not actually take place, but it is a useful mental model as we think about broadcasting.

我们可以认为上面的运算首先将标量扩展成了一个一维的数组`[5, 5, 5]`，然后在和`a`进行了加法运算。NumPy的广播方式并不是真的需要将元素复制然后扩展，但是这对于理解广播的运行方式很有帮助。

> We can similarly extend this to arrays of higher dimension. Observe the result when we add a one-dimensional array to a two-dimensional array:

我们可以很简单的将上面的情形推广到更高纬度的数组上。下面我们使用广播将一个一维数组和一个二维数组进行加法运算：

In [4]:
```
M = np.ones((3, 3))
M
```

Out[4]:
```
array([[1., 1., 1.],
       [1., 1., 1.],
       [1., 1., 1.]])
```

In [5]:
```
M + a
```

Out[5]:
```
array([[1., 2., 3.],
       [1., 2., 3.],
       [1., 2., 3.]])
```

> Here the one-dimensional array `a` is stretched, or broadcast across the second dimension in order to match the shape of `M`.

上例中一维数组`a`在第二个维度上进行了扩展或者广播，这样才能符合`M`的形状。

> While these examples are relatively easy to understand, more complicated cases can involve broadcasting of both arrays. Consider the following example:

上面两个例子相对来说非常容易理解，但是当参与运算的两个数组都需要广播时，情况就相对复杂一些了。看下面的例子：

In [6]:
```
a = np.arange(3)
b = np.arange(3)[:, np.newaxis]

print(a)
print(b)
```

```
[0 1 2]
[[0]
 [1]
 [2]]
```

In [7]:
```
a + b
```

Out[7]:
```
array([[0, 1, 2],
       [1, 2, 3],
       [2, 3, 4]])
```

> Just as before we stretched or broadcasted one value to match the shape of the other, here we've stretched *both* `a` and `b` to match a common shape, and the result is a two-dimensional array! The geometry of these examples is visualized in the following figure (Code to produce this plot can be found in the appendix, and is adapted from source published in the astroML documentation. Used by permission).

前面例子中我们只对其中一个数组进行了扩展或者广播，上例中我们需要对`a`和`b`两个数组都进行广播才能满足双方是相同的形状，最后的结果是一个二维的数组。上面例子可以用下面的图来进行说明（生成这些图像的代码可以在附录中找到，其中部分使用了经过授权的astroML网站文档中的代码）。

Broadcasting Visual

> The light boxes represent the broadcasted values: again, this extra memory is not actually allocated in the course of the operation, but it can be useful conceptually to imagine that it is.

浅色格子代表的是广播后的值：再次说明，这些广播的值不会真正占用内存，只是为了辅助我们理解广播的机制。

Rules of Broadcasting

广播的规则

> Broadcasting in NumPy follows a strict set of rules to determine the interaction between the two arrays:
>
> - Rule 1: If the two arrays differ in their number of dimensions, the shape of the one with fewer dimensions is *padded* with ones on its leading (left) side.
> - Rule 2: If the shape of the two arrays does not match in any dimension, the array with shape equal to 1 in that dimension is stretched to match the other shape.
> - Rule 3: If in any dimension the sizes disagree and neither is equal to 1, an error is raised.

在NumPy中应用广播不是随意的，而是需要遵从严格的一套规则：

- 规则1：如果两个数组有着不同的维度，维度较小的那个数组会沿着最前（或最左）的维度进行扩增，扩增的维度尺寸为1，这时两个数组具有相同的维度。
- 规则2：如果两个数组形状在任何某个维度上存在不相同，那么两个数组中形状为1的维度都会广播到另一个数组对应唯独的尺寸，最终双方都具有相同的形状。
- 规则3：如果两个数组在同一个维度上具有不为1的不同长度，那么将产生一个错误。

> To make these rules clear, let's consider a few examples in detail.

为了说明白这些规则，我们需要参考下面的一些例子：

Broadcasting example 1

广播规则例子1

> Let's look at adding a two-dimensional array to a one-dimensional array:

我们先看一下一个二维数组和一个一维数组相加：

In [8]:
```
M = np.ones((2, 3))
a = np.arange(3)
```

> Let's consider an operation on these two arrays. The shape of the arrays are

我们先看一下两个数组的形状：

- `M.shape = (2, 3)`
- `a.shape = (3,)`

> We see by rule 1 that the array `a` has fewer dimensions, so we pad it on the left with ones:

依据规则1，数组`a`的维度较少，因此首先对其进行维度扩增，我们在其最前面（最左边）增加一个维度，长度为1。此时两个数组的形状变为：

- `M.shape -> (2, 3)`
- `a.shape -> (1, 3)`

> By rule 2, we now see that the first dimension disagrees, so we stretch this dimension to match:

依据规则2，我们可以看到双方在第一维度上不相同，因此我们将第一维度具有长度1的`a`的第一维度扩展为2。此时双方的形状变为：

- `M.shape -> (2, 3)`
- `a.shape -> (2, 3)`

> The shapes match, and we see that the final shape will be `(2, 3)`:

经过变换之后，双方形状一致，可以进行加法运算了，我们可以预知最终结果的形状为`(2, 3)`：

In [9]:
```
M + a
```

Out[9]:
```
array([[1., 2., 3.],
       [1., 2., 3.]])
```

Broadcasting example 2

广播规则例子2

> Let's take a look at an example where both arrays need to be broadcast:

让我们看一个两个数组都需要广播的情况：

In [10]:
```
a = np.arange(3).reshape((3, 1))
b = np.arange(3)
```

> Again, we'll start by writing out the shape of the arrays:

开始时双方的形状为：

- `a.shape = (3, 1)`
- `b.shape = (3,)`

> Rule 1 says we must pad the shape of `b` with ones:

由规则1我们需要将数组`b`扩增第一维度，长度为1：

- `a.shape -> (3, 1)`
- `b.shape -> (1, 3)`

> And rule 2 tells us that we upgrade each of these ones to match the corresponding size of the other array:

由规则2我们需要将数组`a`的第二维度扩展为3，还需要将数组`b`的第一维度扩展为3，得到：

- `a.shape -> (3, 3)`
- `b.shape -> (3, 3)`

> Because the result matches, these shapes are compatible. We can see this here:

双方形状相同，可以进行运算：

In [11]:
```
a + b
```

Out[11]:
```
array([[0, 1, 2],
       [1, 2, 3],
       [2, 3, 4]])
```

Broadcasting example 3

广播规则例子3

> Now let's take a look at an example in which the two arrays are not compatible:

现在我们来看一个不能适用于广播的例子：

In [12]:
```
M = np.ones((3, 2))
a = np.arange(3)
```

> This is just a slightly different situation than in the first example: the matrix `M` is transposed. How does this affect the calculation? The shape of the arrays are

这个例子和例子1有一点点区别，那就是本例中的`M`是例子1中`M`的转置矩阵。它们的形状是：

- `M.shape = (3, 2)`
- `a.shape = (3,)`

> Again, rule 1 tells us that we must pad the shape of `a` with ones:

由规则1我们需要在数组`a`上扩增第一维度，长度为1：

- `M.shape -> (3, 2)`
- `a.shape -> (1, 3)`

> By rule 2, the first dimension of `a` is stretched to match that of `M`:

由规则2我们需要将数组`a`的第一维度扩展为3才能与数组`M`保持一致，除此之外双方都没有长度为1的维度了：

- `M.shape -> (3, 2)`
- `a.shape -> (3, 3)`

> Now we hit rule 3–the final shapes do not match, so these two arrays are incompatible, as we can observe by attempting this operation:

观察得到的形状，你可以发现这个结果满足规则3，双方的各维度长度不完全一致且不为1，因此无法完成广播，最终会产生错误：

In [13]:
```
M + a
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-13-8cac1d547906> in <module>
----> 1 M + a

ValueError: operands could not be broadcast together with shapes (3,2) (3,)
```

> Note the potential confusion here: you could imagine making `a` and `M` compatible by, say, padding `a`'s shape with ones on the right rather than the left. But this is not how the broadcasting rules work! That sort of flexibility might be useful in some cases, but it would lead to potential areas of ambiguity. If right-side padding is what you'd like, you can do this explicitly by reshaping the array (we'll use the `np.newaxis` keyword introduced in The Basics of NumPy Arrays):

这里你可能会发现一个问题：如果广播的时候不一定按照最前面（最左边）维度的原则进行扩增维度的话，那不是很多的数组都可以进行广播计算吗？这样处理不是更灵活吗？例如上例中如果我们在数组`a`的第二维度上扩增的话，那广播就能正确进行了。很可惜，广播并不会支持这种处理方式，虽然这种方法在某些情况下会更加灵活，但是在部分情况下会带来不确定性。如果你确实希望进行右维度扩增的话，你必须明确指定。利用我们在NumPy数组基础中介绍的`np.newaxis`属性可以进行这个操作：

In [14]:
```
a[:, np.newaxis].shape
```

Out[14]:
```
(3, 1)
```

In [15]:
```
M + a[:, np.newaxis]
```

Out[15]:
```
array([[1., 1.],
       [2., 2.],
       [3., 3.]])
```

> Also note that while we've been focusing on the `+` operator here, these broadcasting rules apply to *any* binary `ufunc`. For example, here is the `logaddexp(a, b)` function, which computes `log(exp(a) + exp(b))` with more precision than the naive approach:

还要说明的是，上面的例子中我们都是使用加法进行说明，实际上广播可以应用到*任何*的二元ufunc上。例如下面我们采用`logaddexp(a, b)`函数求值，这个函数计算的是$\log(e^a + e^b)$的值，使用这个函数能比采用原始的exp和log函数进行计算得到更高的精度：

In [16]:
```
np.logaddexp(M, a[:, np.newaxis])
```

Out[16]:
```
array([[1.31326169, 1.31326169],
       [1.69314718, 1.69314718],
       [2.31326169, 2.31326169]])
```

> For more information on the many available universal functions, refer to Computation on NumPy Arrays: Universal Functions.

更多关于通用函数的介绍，请复习使用Numpy计算：通用函数。

Broadcasting in Practice

广播规则实践

> Broadcasting operations form the core of many examples we'll see throughout this book. We'll now take a look at a couple simple examples of where they can be useful.

广播操作在本书后面很多例子中都会见到。因此这里我们看一些简单的例子，更好的说明它。

Centering an array

中心化数组

> In the previous section, we saw that ufuncs allow a NumPy user to remove the need to explicitly write slow Python loops. Broadcasting extends this ability. One commonly seen example is when centering an array of data. Imagine you have an array of 10 observations, each of which consists of 3 values. Using the standard convention (see Data Representation in Scikit-Learn), we'll store this in a 10×3 array:

在前一节中，我们看到了ufuncs提供了我们可以避免使用Python循环的低效方式，而广播则大大扩展了这种能力。一个常见的例子就是我们需要将数据集进行中心化。例如我们我们进行了10次采样观测，每次都会得到3个数据值。按照惯例（参见Scikit-Learn数据表现方式），我们可以将这些数据存成一个10×3的数组：

In [17]:
```
X = np.random.random((10, 3))
```

> We can compute the mean of each feature using the `mean` aggregate across the first dimension:

我们使用`mean`函数沿着第一维度求出每个特征的平均值：

In [18]:
```
Xmean = X.mean(0)
Xmean
```

Out[18]:
```
array([0.61839754, 0.51852053, 0.65514576])
```

> And now we can center the `X` array by subtracting the mean (this is a broadcasting operation):

下面我们就可以将数组`X`减去它的各维度平均值就可以将其中心化（这里就是一个广播操作）：

In [19]:
```
X_centered = X - Xmean
```

> To double-check that we've done this correctly, we can check that the centered array has near zero mean:

我们来检查一下结果的正确性，我们可以通过查看中心化后的数组在各特征上的平均值是够接近于0来进行判断：

In [20]:
```
X_centered.mean(0)
```

Out[20]:
```
array([-1.11022302e-17, -7.77156117e-17,  6.66133815e-17])
```

> To within machine precision, the mean is now zero.

考虑到机器精度情况，平均值已经等于0了。

Plotting a two-dimensional function

绘制二维函数的图形

> One place that broadcasting is very useful is in displaying images based on two-dimensional functions. If we want to define a function $z = f(x, y)$, broadcasting can be used to compute the function across the grid:

广播还有一个很有用的场景，就是当你需要绘制一个二维函数的图像时。如果我们希望定义一个函数$z = f(x, y)$，广播可以被用来计算二维平面上每个网格的数值：

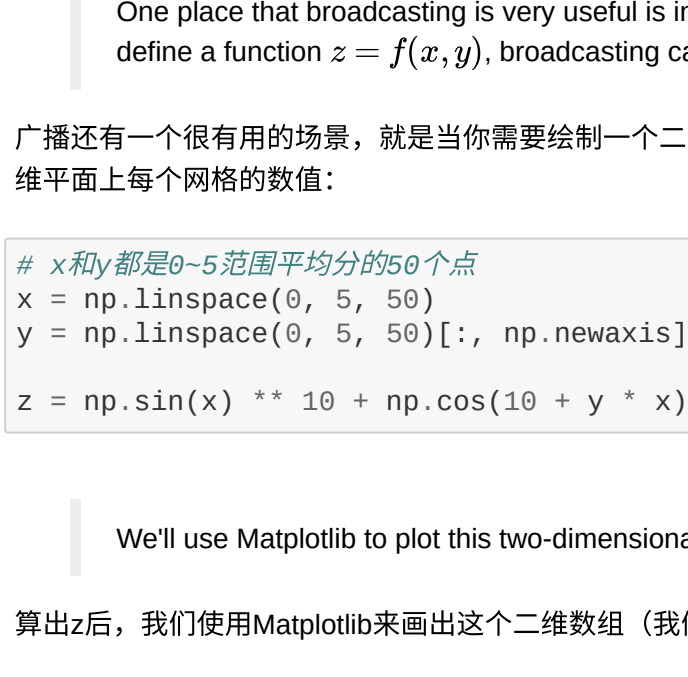
In [21]:
```
# x和y都是0~5范围平均分的50个点
x = np.linspace(0, 5, 50)
y = np.linspace(0, 5, 50)[:, np.newaxis]

z = np.sin(x) ** 10 + np.cos(10 + y * x) * np.cos(x)
```

> We'll use Matplotlib to plot this two-dimensional array (these tools will be discussed in full in Density and Contour Plots):

算出z后，我们使用Matplotlib来画出这个二维数组（我们将在密度和轮廓图中详细介绍）：

In [22]:
```
%matplotlib inline
import matplotlib.pyplot as plt
```

In [23]:
```
plt.imshow(z, origin='lower', extent=[0, 5, 0, 5],
           cmap='viridis')
plt.colorbar();
```



> The result is a compelling visualization of the two-dimensional function.

上面的图形以一种极其吸引人的方式为我们展现了二维函数的分布情况。