```
In[13]: M + a
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-13-9e16e9f98da6> 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" on page 42):

```
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:

```
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" on page 50.

Broadcasting in Practice

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.

Centering an array

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 com-

monly seen example is 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" on page 343), we'll store this in a 10×3 array:

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

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

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

Out[18]: array([ 0.53514715,  0.66567217,  0.44385899])
```

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

```
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:

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

Out[20]: array([  2.22044605e-17,  -7.77156117e-17,  -1.66533454e-17])
```

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

Plotting a two-dimensional function

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:

```
In[21]: # x and y have 50 steps from 0 to 5
        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" on page 241):

```
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, shown in Figure 2-5, is a compelling visualization of the two-dimensional function.

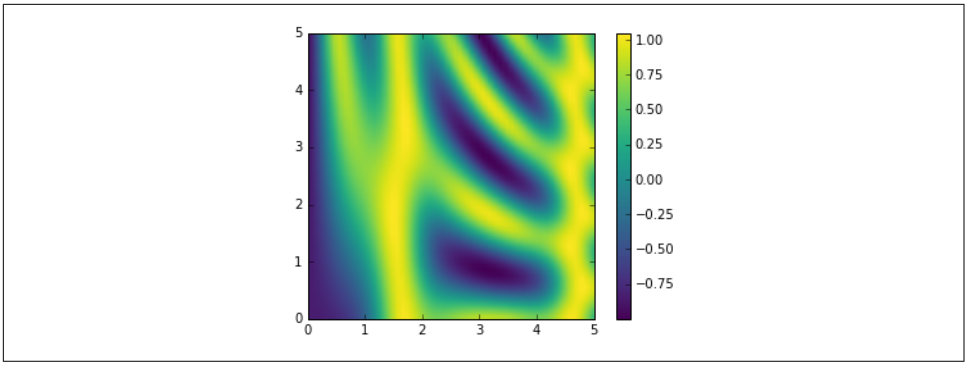

Figure 2-5. Visualization of a 2D array

Comparisons, Masks, and Boolean Logic

This section covers the use of Boolean masks to examine and manipulate values within NumPy arrays. Masking comes up when you want to extract, modify, count, or otherwise manipulate values in an array based on some criterion: for example, you might wish to count all values greater than a certain value, or perhaps remove all outliers that are above some threshold. In NumPy, Boolean masking is often the most efficient way to accomplish these types of tasks.

Example: Counting Rainy Days

Imagine you have a series of data that represents the amount of precipitation each day for a year in a given city. For example, here we'll load the daily rainfall statistics for the city of Seattle in 2014, using Pandas (which is covered in more detail in Chapter 3):

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

       # use Pandas to extract rainfall inches as a NumPy array
       rainfall = pd.read_csv('data/Seattle2014.csv')['PRCP'].values
       inches = rainfall / 254  # 1/10mm -> inches
       inches.shape
```

```
Out[1]: (365,)
```

The array contains 365 values, giving daily rainfall in inches from January 1 to December 31, 2014.

As a first quick visualization, let's look at the histogram of rainy days shown in Figure 2-6, which was generated using Matplotlib (we will explore this tool more fully in Chapter 4):