```
# Plot the result as an image
plt.imshow(Z.reshape(Xgrid.shape),
           origin='lower', aspect='auto',
           extent=[-3.5, 3.5, -6, 6],
           cmap='Blues')
cb = plt.colorbar()
cb.set_label("density")
```

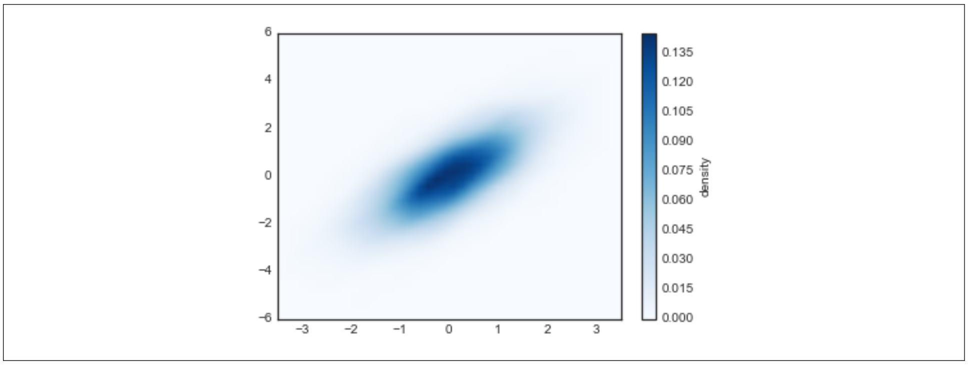

Figure 4-40. A kernel density representation of a distribution

KDE has a smoothing length that effectively slides the knob between detail and smoothness (one example of the ubiquitous bias–variance trade-off). The literature on choosing an appropriate smoothing length is vast: `gaussian_kde` uses a rule of thumb to attempt to find a nearly optimal smoothing length for the input data.

Other KDE implementations are available within the SciPy ecosystem, each with its own various strengths and weaknesses; see, for example, `sklearn.neighbors.KernelDensity` and `statsmodels.nonparametric.kernel_density.KDEMultivariate`. For visualizations based on KDE, using Matplotlib tends to be overly verbose. The Seaborn library, discussed in "Visualization with Seaborn" on page 311, provides a much more terse API for creating KDE-based visualizations.

Customizing Plot Legends

Plot legends give meaning to a visualization, assigning labels to the various plot elements. We previously saw how to create a simple legend; here we'll take a look at customizing the placement and aesthetics of the legend in Matplotlib.

The simplest legend can be created with the `plt.legend()` command, which automatically creates a legend for any labeled plot elements (Figure 4-41):

```
In[1]: import matplotlib.pyplot as plt
       plt.style.use('classic')
```

```
In[2]: %matplotlib inline
       import numpy as np
```

```
In[3]: x = np.linspace(0, 10, 1000)
       fig, ax = plt.subplots()
       ax.plot(x, np.sin(x), '-b', label='Sine')
       ax.plot(x, np.cos(x), '--r', label='Cosine')
       ax.axis('equal')
       leg = ax.legend();
```

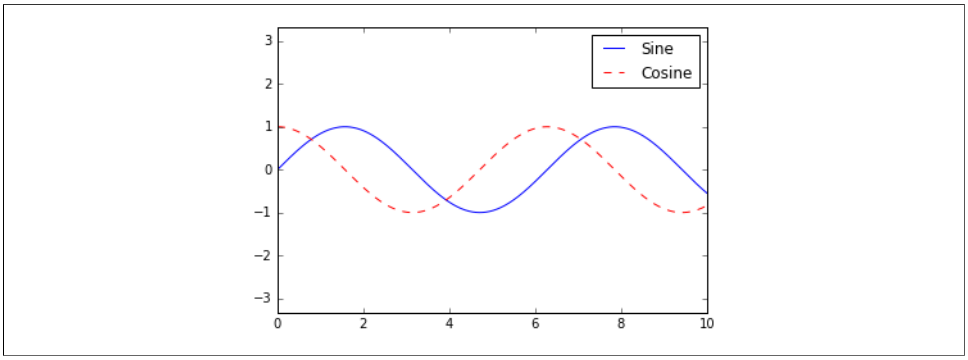

Figure 4-41. A default plot legend

But there are many ways we might want to customize such a legend. For example, we can specify the location and turn off the frame (Figure 4-42):

```
In[4]: ax.legend(loc='upper left', frameon=False)
       fig
```

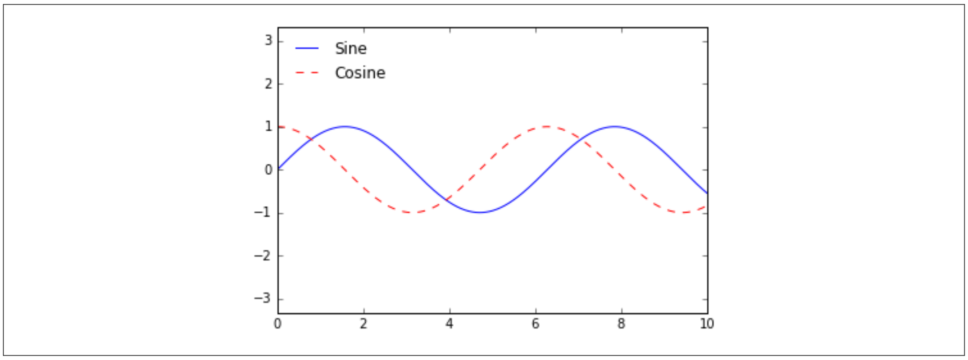

Figure 4-42. A customized plot legend

We can use the `ncol` command to specify the number of columns in the legend (Figure 4-43):

```
In[5]: ax.legend(frameon=False, loc='lower center', ncol=2)
       fig
```

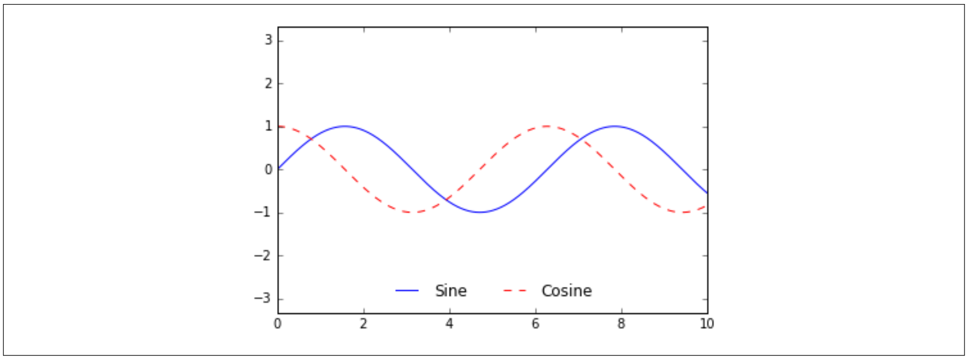

Figure 4-43. A two-column plot legend

We can use a rounded box (`fancybox`) or add a shadow, change the transparency (alpha value) of the frame, or change the padding around the text (Figure 4-44):

```
In[6]: ax.legend(fancybox=True, framealpha=1, shadow=True, borderpad=1)
       fig
```

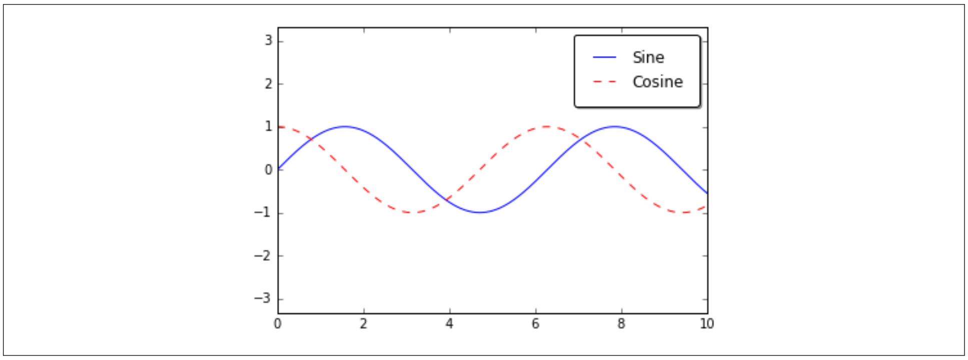

Figure 4-44. A fancybox plot legend

For more information on available legend options, see the `plt.legend` docstring.

Choosing Elements for the Legend

As we've already seen, the legend includes all labeled elements by default. If this is not what is desired, we can fine-tune which elements and labels appear in the legend by using the objects returned by plot commands. The `plt.plot()` command is able to create multiple lines at once, and returns a list of created line instances. Passing any of these to `plt.legend()` will tell it which to identify, along with the labels we'd like to specify (Figure 4-45):

```
In[7]: y = np.sin(x[:, np.newaxis] + np.pi * np.arange(0, 2, 0.5))
       lines = plt.plot(x, y)
```