Lecture 14 : Data Visualization (Part 2)

Data Science, DST, UIC

Table of content of this chapter:

- Histograms
- Customizing Plot Legends

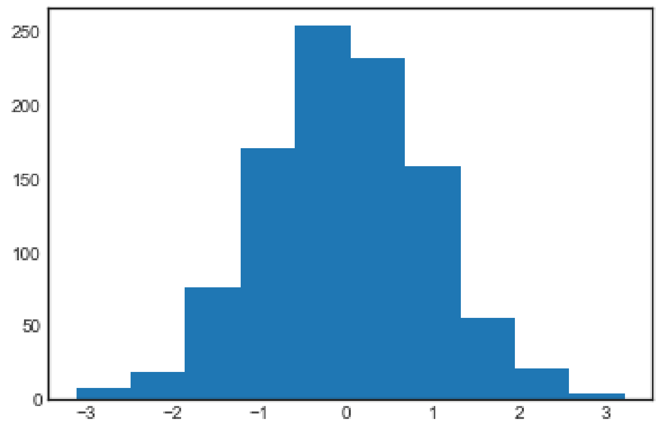
Histograms

A simple histogram can be a great first step in understanding a dataset.

In [1]:
```
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
plt.style.use('seaborn-white')

data = np.random.randn(1000)
```

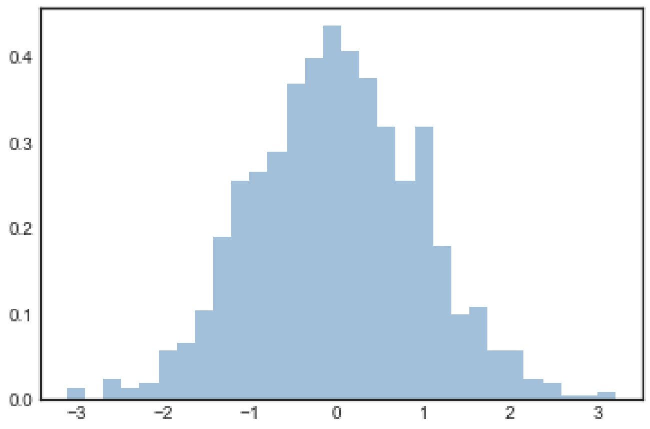
In [2]:
```
plt.hist(data);
```



The `hist(x)` function has many options to tune both the calculation and the display.

- `x` : (n,) array or sequence of (n,) arrays
- `bins` : int or sequence or str, optional. If an integer is given, `bins + 1` bin edges are calculated and returned.
- `density` : counts will be normalized. This is achieved by dividing the count by the number of observations.
- `histtype` : {'bar', 'barstacked', 'step', 'stepfilled'}, optional. The type of histogram to draw.

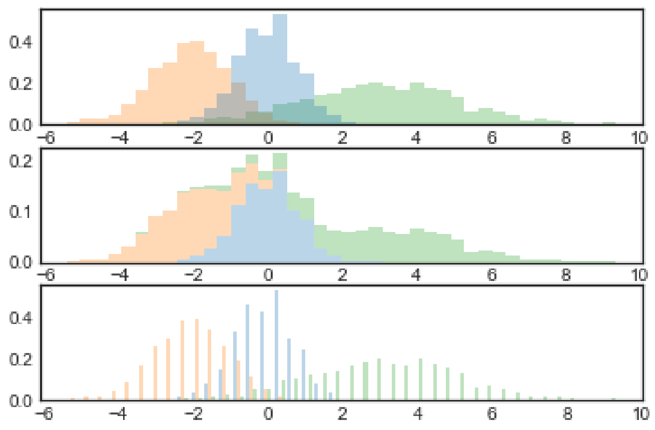
In [3]:
```
plt.hist(data, bins=30, density=True, alpha=0.5,
         histtype='stepfilled', color='steelblue',
         edgecolor='none');
```



The `plt.hist` docstring has more information on other customization options available. I find this combination of `histtype='stepfilled'` along with some transparency `alpha` to be very useful when comparing histograms of several distributions:

In [4]:
```
x1 = np.random.normal(0, 0.8, 1000)
x2 = np.random.normal(-2, 1, 1000)
x3 = np.random.normal(3, 2, 1000)

fig,ax = plt.subplots(3,1)
kwargs = dict(alpha=0.3, density=True, bins=40)
ax[0].hist([x1,x2,x3], histtype='stepfilled', **kwargs)
ax[1].hist([x1,x2,x3], histtype='barstacked', **kwargs)
ax[2].hist([x1,x2,x3], histtype='bar',**kwargs);
```



If you would like to simply compute the histogram (that is, count the number of points in a given bin) and not display it, the `np.histogram()` function is available:

In [5]:
```
counts, bin_edges = np.histogram(data, bins=5)
print(counts)
print(bin_edges)
```

```
[ 27 247 486 215  25]
[-3.12116295 -1.85416893 -0.58717491  0.67981911  1.94681314  3.21380716]
```

Two-Dimensional Histograms and Binnings

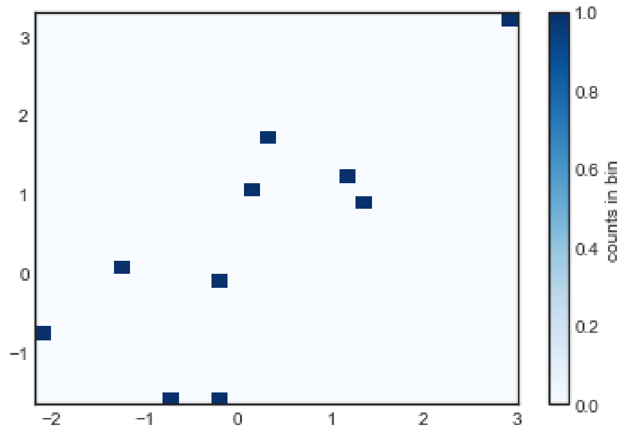
Just as we create histograms in one dimension by dividing the number-line into bins, we can also create histograms in two-dimensions by dividing points among two-dimensional bins. We'll take a brief look at several ways to do this here. We'll start by defining some data—an `x` and `y` array drawn from a multivariate Gaussian distribution:

In [6]:
```
mean = [0, 0]
cov = [[1, 1], [1, 2]]
x, y = np.random.multivariate_normal(mean, cov, 10).T
```

`plt.hist2d`: Two-dimensional histogram

One straightforward way to plot a two-dimensional histogram is to use Matplotlib's `plt.hist2d` function:

In [7]:
```
plt.hist2d(x, y, bins=30, cmap='Blues')
cb = plt.colorbar()
cb.set_label('counts in bin')
```



Just as with `plt.hist`, `plt.hist2d` has a number of extra options to fine-tune the plot and the binning, which are nicely outlined in the function docstring. Further, just as `plt.hist` has a counterpart in `np.histogram`, `plt.hist2d` has a counterpart in `np.histogram2d`, which can be used as follows:

In [8]:
```
counts, xedges, yedges = np.histogram2d(x, y, bins=30)
counts.shape
```

Out[8]:
```
(30, 30)
```

In [9]:
```
xedges.shape
```

Out[9]:
```
(31,)
```

In [10]:
```
yedges.shape
```

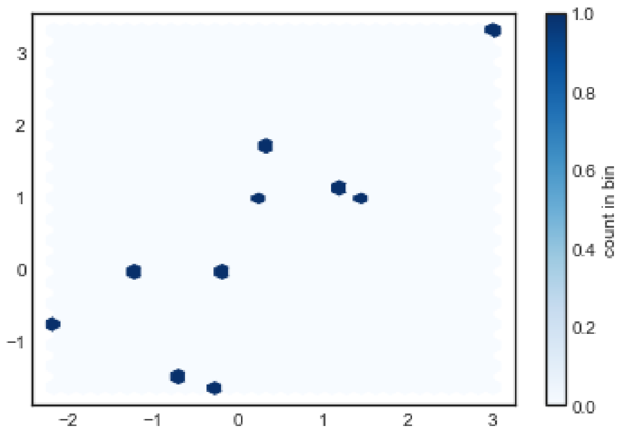
Out[10]:
```
(31,)
```

For the generalization of this histogram binning in dimensions higher than two, see the `np.histogramdd` function.

`plt.hexbin`: Hexagonal binnings

The two-dimensional histogram creates a tesselation of squares across the axes. Another natural shape for such a tesselation is the regular hexagon. For this purpose, Matplotlib provides the `plt.hexbin` routine, which will represents a two-dimensional dataset binned within a grid of hexagons:

In [11]:
```
plt.hexbin(x, y, gridsize=30, cmap='Blues')
cb = plt.colorbar(label='count in bin')
```



`plt.hexbin` has a number of interesting options, including the ability to specify weights for each point, and to change the output in each bin to any NumPy aggregate (mean of weights, standard deviation of weights, etc.).

Customizing Plot Legends

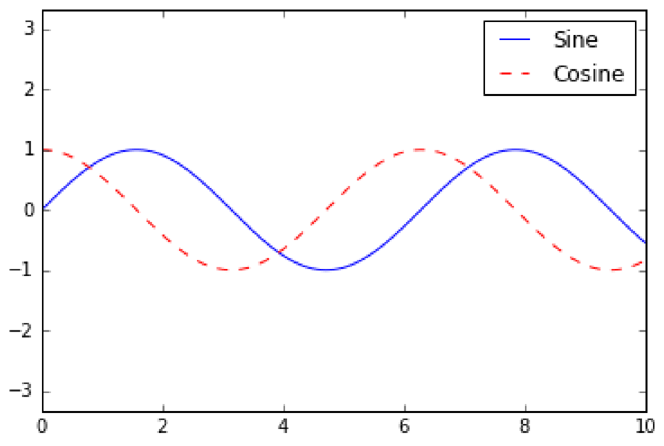
Plot legends give meaning to a visualization, assigning meaning to the various plot elements. We previously saw how to create a simple legend; here we'll take a look at customizing the placement and aesthetics of the legend in Matplotlib.

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

In [12]:
```
import matplotlib.pyplot as plt
plt.style.use('classic')
```

In [13]:
```
%matplotlib inline
import numpy as np
```

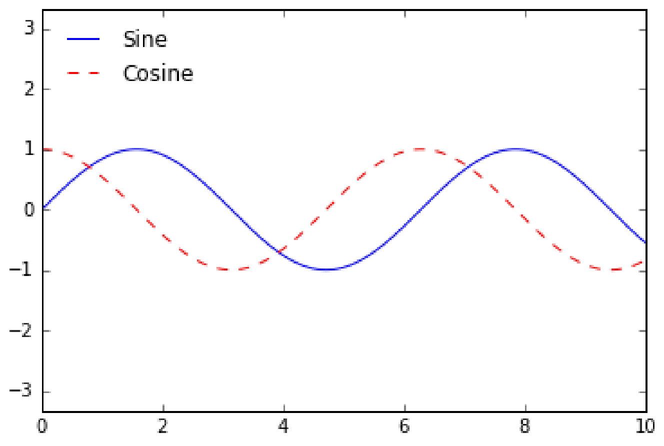
In [14]:
```
x = np.linspace(0, 10, 1000)
fig, ax = plt.subplots()
ax.plot(x, np.sin(x), '-b', label='Sine')
ax.plot(x, np.cos(x), '--r', label='Cosine')
ax.axis('equal')
ax.legend();
```



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:

In [15]:
```
ax.legend(loc='upper left', frameon=False)
fig
```

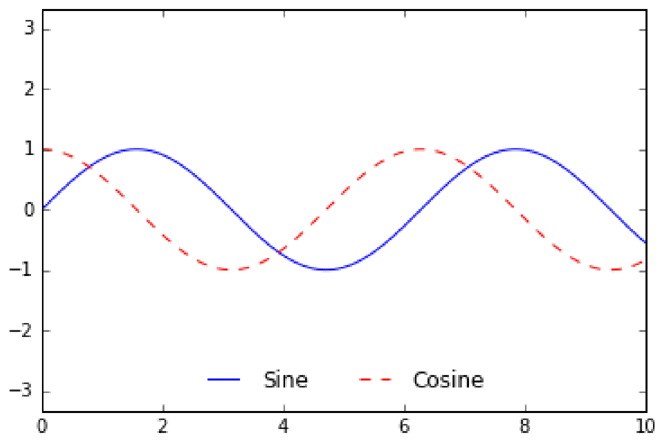
Out[15]:



We can use the `ncol` command to specify the number of columns in the legend:

In [16]:
```
ax.legend(frameon=False, loc='lower center', ncol=2)
fig
```

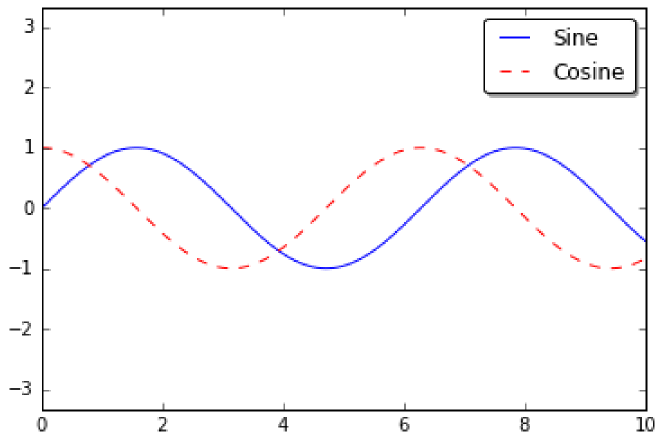
Out[16]:



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:

In [17]:
```
ax.legend(fancybox=True, framealpha=1, shadow=True)
fig
```

Out[17]:



Notice that by default, the legend ignores all elements without a `label` attribute set. For more information on available legend options, see the `plt.legend` docstring.

Legend for Size of Points

Sometimes the legend defaults are not sufficient for the given visualization. For example, perhaps you're be using the size of points to mark certain features of the data, and want to create a legend reflecting this. Here is an example where we'll use the size of points to indicate populations of California cities. We'd like a legend that specifies the scale of the sizes of the points, and we'll accomplish this by plotting some labeled data with no entries:

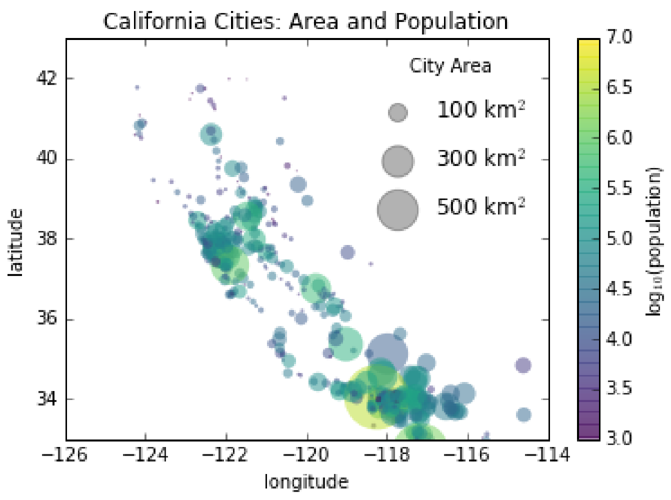
In [21]:
```
import pandas as pd
cities = pd.read_csv('california_cities.csv')

# Extract the data we're interested in
lat, lon = cities['latd'], cities['longd']
population, area = cities['population_total'], cities['area_total']

# Scatter the points, using size and color but no label
plt.scatter(lon, lat, label=None,
            c=np.log10(population), cmap='viridis',
            s=area, linewidth=0, alpha=0.5)
plt.axis('equal')
plt.xlabel('longitude')
plt.ylabel('latitude')
plt.colorbar(label='log$_{10}$(population)')
plt.clim(3, 7)

# Here we create a legend:
# we'll plot empty lists with the desired size and label
for area in [100, 300, 500]:
    plt.scatter([], [], c='k', alpha=0.3, s=area,
                label=str(area) + ' km$^2$')
plt.legend(scatterpoints=1, frameon=False, labelspacing=1, title='City Area')

plt.title('California Cities: Area and Population');
```



In [19]:
```
plt.legend?
```

The legend will always reference some object that is on the plot, so if we'd like to display a particular shape we need to plot it. In this case, the objects we want (gray circles) are not on the plot, so we fake them by plotting empty lists. Notice too that the legend only lists plot elements that have a label specified.

By plotting empty lists, we create labeled plot objects which are picked up by the legend, and now our legend tells us some useful information. This strategy can be useful for creating more sophisticated visualizations.

Multiple Legends

Sometimes when designing a plot you'd like to add multiple legends to the same axes. Unfortunately, Matplotlib does not make this easy: via the standard `legend` interface, it is only possible to create a single legend for the entire plot. If you try to create a second legend using `plt.legend()` or `ax.legend()` , it will simply override the first one. We can work around this by creating a new legend artist from scratch, and then using the lower-level `ax.add_artist()` method to manually add the second artist to the plot:

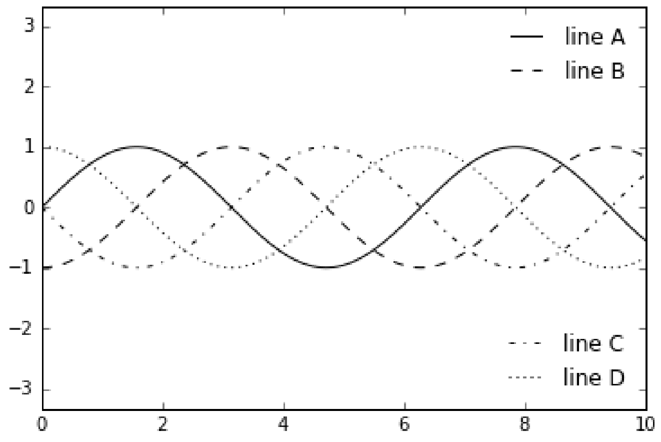
In [20]:
```
fig, ax = plt.subplots(1,1)

lines = []
styles = ['-', '--', '-.', ':']
x = np.linspace(0, 10, 1000)

for i in range(4):
    lines += ax.plot(x, np.sin(x - i * np.pi / 2),
                     styles[i], color='black')
ax.axis('equal')

# specify the lines and labels of the first legend
ax.legend(lines[:2], ['line A', 'line B'],
          loc='upper right', frameon=False)

# Create the second legend and add the artist manually.
from matplotlib.legend import Legend
leg = Legend(ax, lines[2:], ['line C', 'line D'],
             loc='lower right', frameon=False)
ax.add_artist(leg);
```



This is a peek into the low-level artist objects that comprise any Matplotlib plot. If you examine the source code of `ax.legend()` (recall that you can do this with within the IPython notebook using `ax.legend??`) you'll see that the function simply consists of some logic to create a suitable `Legend` artist, which is then saved in the `legend_` attribute and added to the figure when the plot is drawn.