VISUALIZATION WITH MATPLOTLIB

General Matplotlib Tips

Importing matplotlib

```
import matplotlib as mpl
import matplotlib.pyplot as plt
```

Setting Styles

```
plt.style.use('classic’)
```

show() or No show()? How to Display Your Plots

Plotting from a script

```
# ------- file: myplot.py ------
import matplotlib.pyplot as plt
import numpy as np
x = np.linspace(0, 10, 100)
plt.plot(x, np.sin(x))
plt.plot(x, np.cos(x))
plt.show()
```

Plotting from an IPython notebook

The IPython notebook is a browser-based interactive data analysis tool that can combine narrative, code, graphics, HTML elements, and much more into a single executable Document

Plotting interactively within an IPython notebook can be done with the %matplotlib command, and works in a similar way to the IPython shell. In the IPython notebook, you also have the option of embedding graphics directly in the notebook, with two possible options:

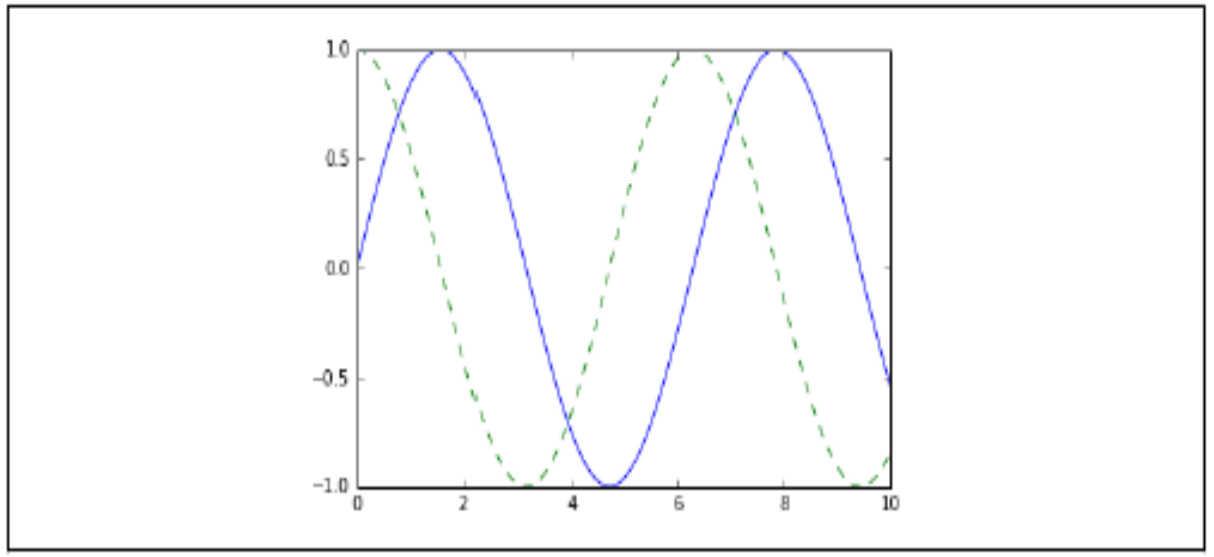
• %matplotlib notebook will lead to *interactive* plots embedded within the notebook

• %matplotlib inline will lead to *static* images of your plot embedded in the notebook

```
%matplotlib inline
```

BASIC PLOTTING EXAMPLE

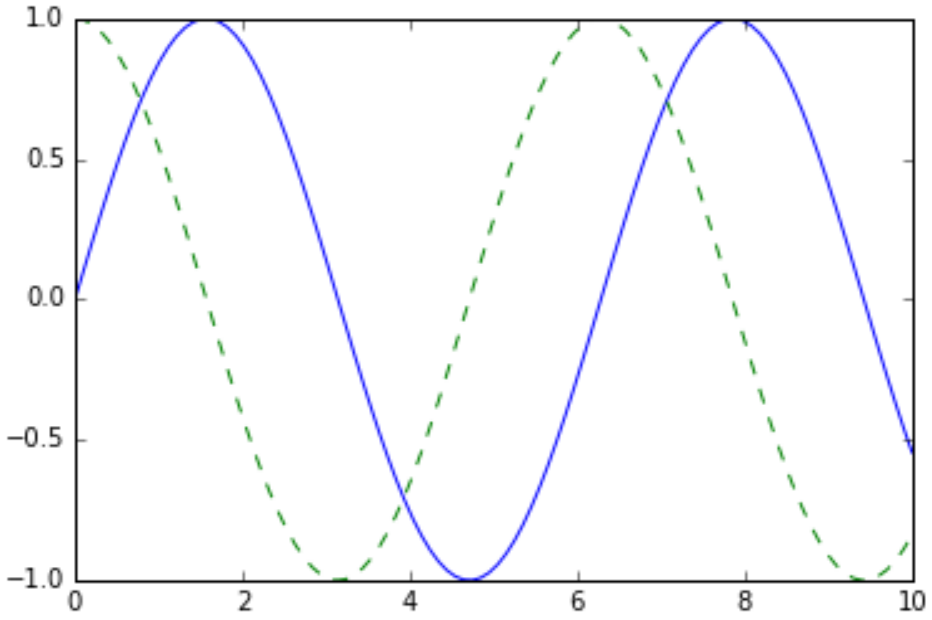
```
import numpy as np
x = np.linspace(0, 10, 100)
fig = plt.figure()
plt.plot(x, np.sin(x), '-')
plt.plot(x, np.cos(x), '--');
```



Saving Figures to File

```
fig.savefig('my_figure.png’)
```

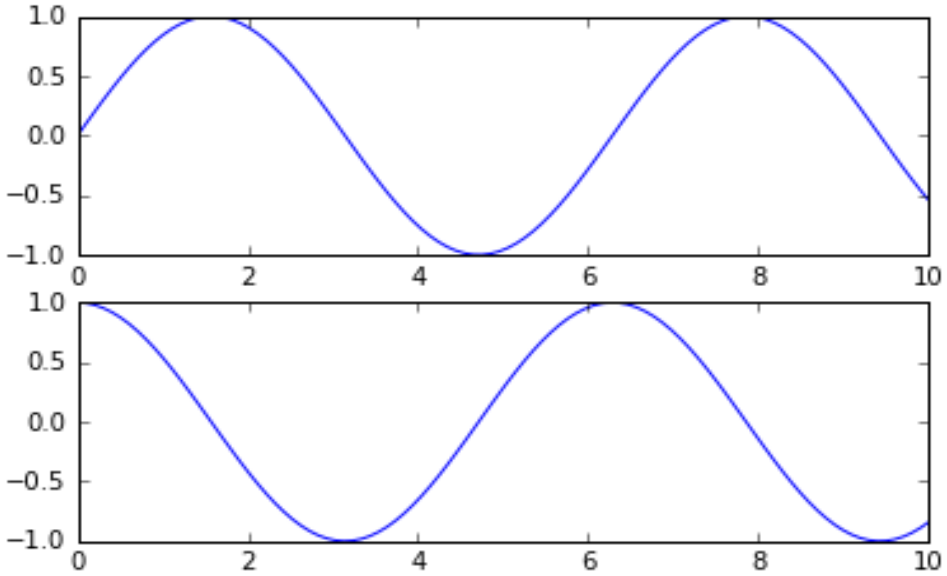
```
from IPython.display import Image
Image('my_figure.png’)
```



Two Interfaces for the Price of One

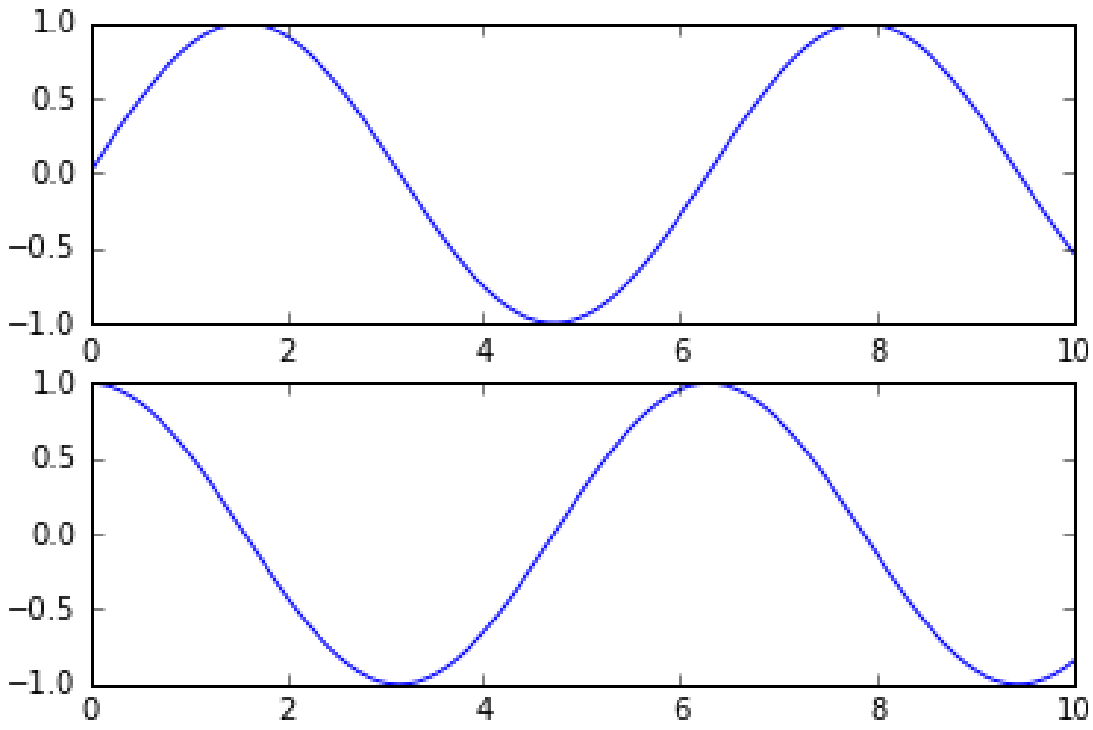
MATLAB-style interface

```
plt.figure() # create a plot figure
# create the first of two panels and set current axis
plt.subplot(2, 1, 1) # (rows, columns, panel number)
plt.plot(x, np.sin(x))
# create the second panel and set current axis
plt.subplot(2, 1, 2)
plt.plot(x, np.cos(x));
```



Object-oriented interface

```
First create a grid of plots
# ax will be an array of two Axes objects
fig, ax = plt.subplots(2)
# Call plot() method on the appropriate object
ax[0].plot(x, np.sin(x))
ax[1].plot(x, np.cos(x));
```



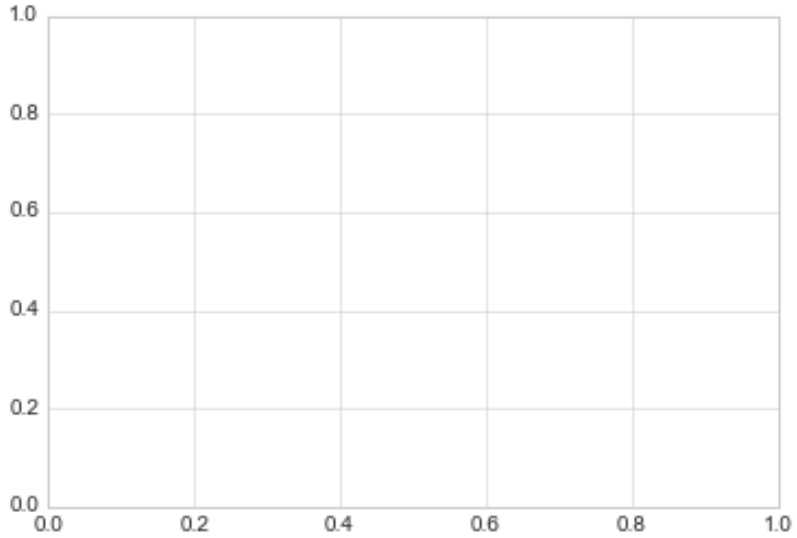
Subplots using the object-oriented interface

SIMPLE LINE PLOTS

The simplest of all plots is the visualization of a single function *y = f (X)*

```
%matplotlib inline
import matplotlib.pyplot as plt
plt.style.use('seaborn-whitegrid')
import numpy as np


fig = plt.figure()
ax = plt.axes()
```



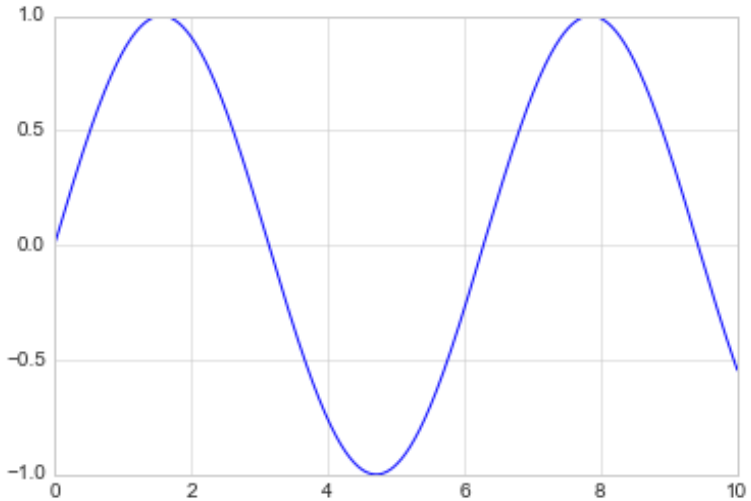
The *axes* (an instance of the class plt.Axes) is what we see above: a bounding

box with ticks and labels, which will eventually contain the plot elements that make

up our visualization. we have created an axes, we can use the ax.plot function to plot some data.

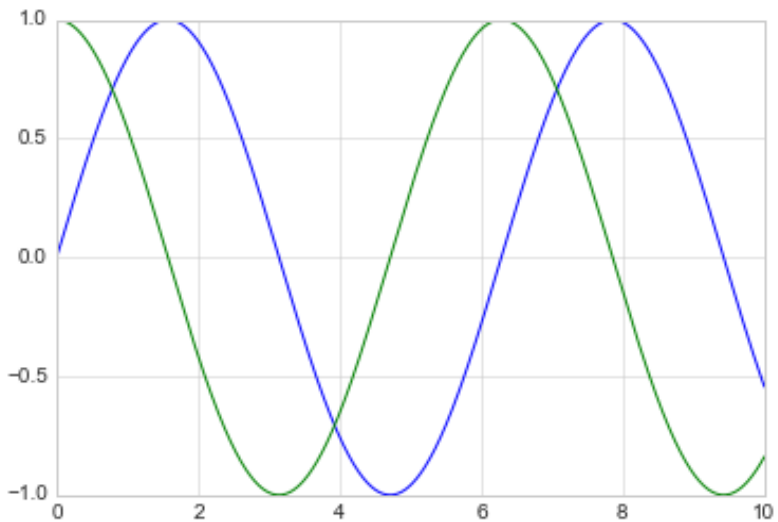
Simple Sinusoidal

```
fig = plt.figure()
ax = plt.axes()
x = np.linspace(0, 10, 1000)
ax.plot(x, np.sin(x));
```



A simple sinusoid via the object-oriented interface

```
plt.plot(x, np.sin(x))
plt.plot(x, np.cos(x));
```



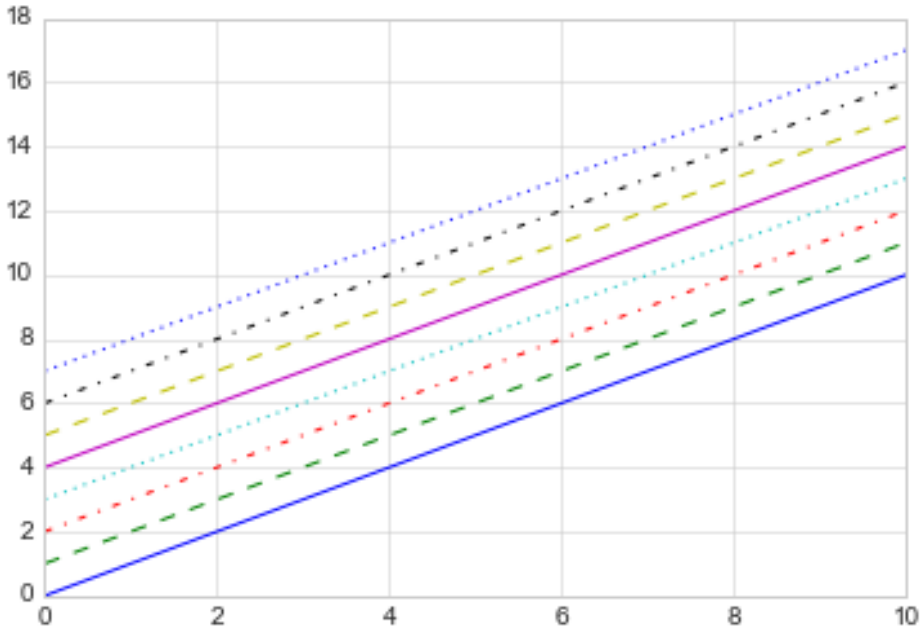
Adjusting the Plot: Line Colors and Styles

The first adjustment you might wish to make to a plot is to control the line colors and styles. The plt.plot() function takes additional arguments that can be used to specify these. To adjust the color, you can use the color keyword, which accepts a string argument representing virtually any imaginable color. plt.plot(x, np.sin(x - 0), color='blue') *# specify color by name*

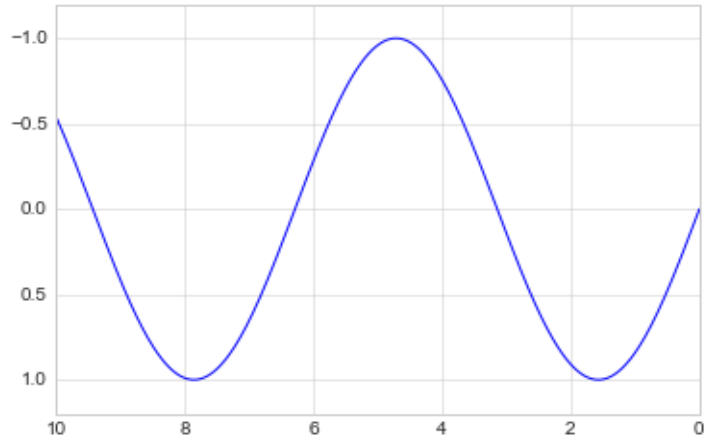
```
plt.plot(x, np.sin(x - 1), color='g') # short color code (rgbcmyk)
plt.plot(x, np.sin(x - 2), color='0.75') # Grayscale between 0 and 1
plt.plot(x, np.sin(x - 3), color='#FFDD44') # Hex code (RRGGBB from 00 to FF)
plt.plot(x, np.sin(x - 4), color=(1.0,0.2,0.3)) # RGB tuple, values 0 and 1
plt.plot(x, np.sin(x - 5), color='chartreuse'); # all HTML
```

Linestyle keyword

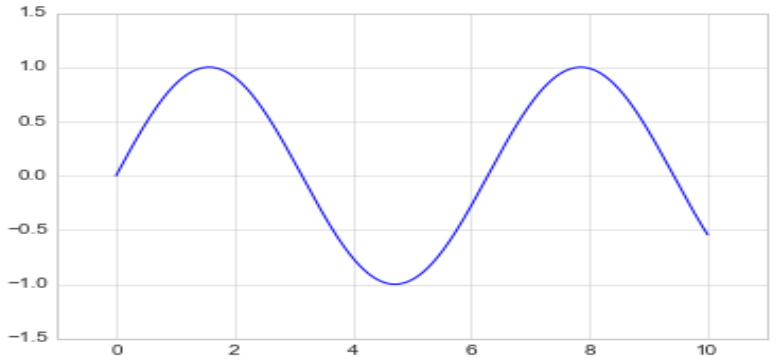
```
plt.plot(x, x + 0, linestyle='solid')
plt.plot(x, x + 1, linestyle='dashed')
plt.plot(x, x + 2, linestyle='dashdot')
plt.plot(x, x + 3, linestyle='dotted');
# For short, you can use the following codes:
plt.plot(x, x + 4, linestyle='-') # solid
plt.plot(x, x + 5, linestyle='--') # dashed
plt.plot(x, x + 6, linestyle='-.') # dashdot
plt.plot(x, x + 7, linestyle=':'); # dotted
```



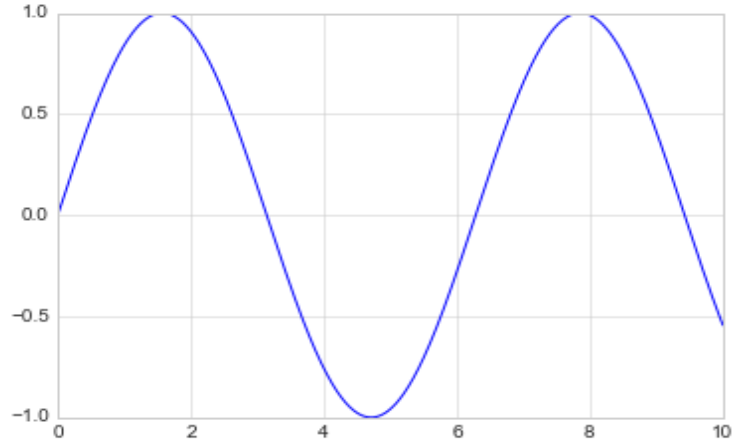
```
plt.plot(x, np.sin(x))
plt.xlim(10, 0)
plt.ylim(1.2, -1.2);
```



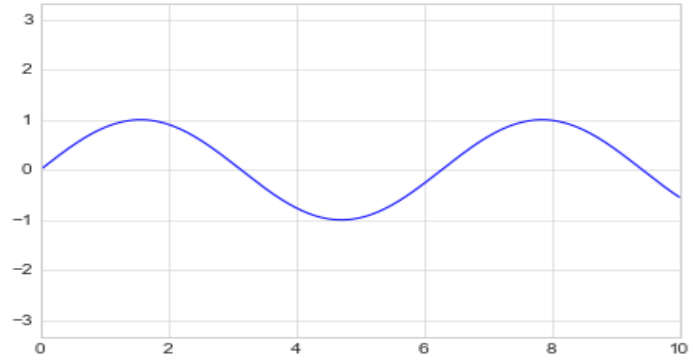

```
plt.plot(x, np.sin(x))
plt.axis([-1, 11, -1.5, 1.5]);
```




```
plt.plot(x, np.sin(x))
plt.axis('tight');
```



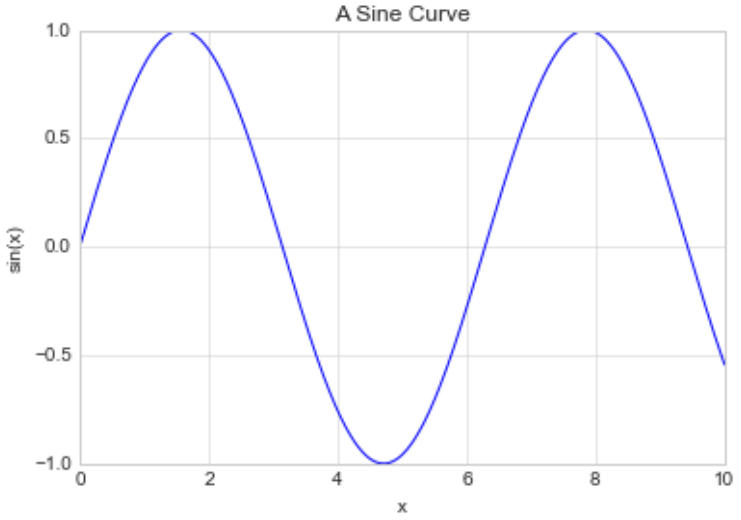
```
plt.plot(x, np.sin(x))
plt.axis('equal');
```



Labeling Plots

```
plt.plot(x, np.sin(x))
plt.title("A Sine Curve")

plt.xlabel("x")
plt.ylabel("sin(x)");
```

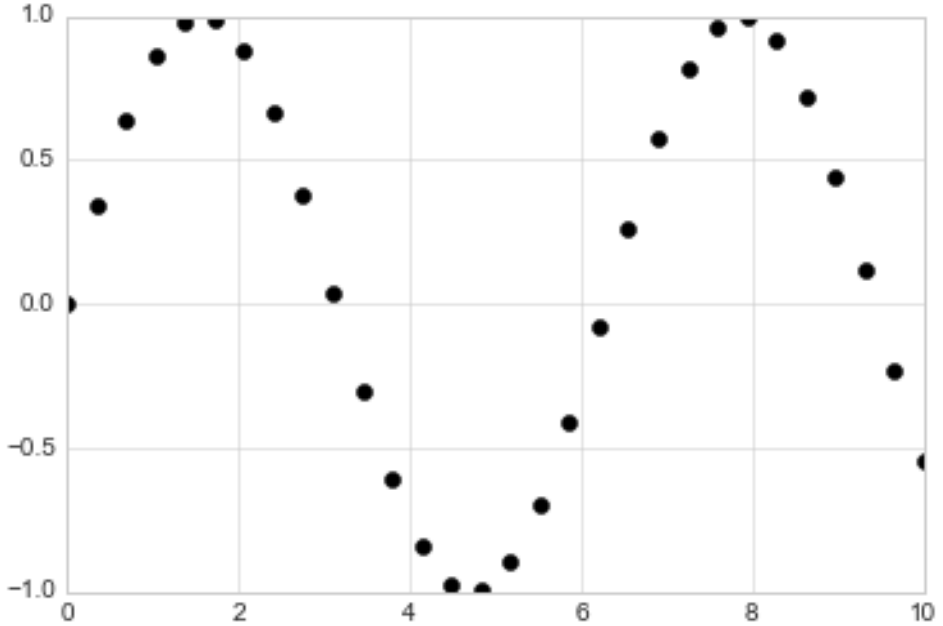


SIMPLE SCATTER PLOTS

```
%matplotlib inline
import matplotlib.pyplot as plt
plt.style.use('seaborn-whitegrid')
import numpy as np
```

Scatter Plots with plt.plot

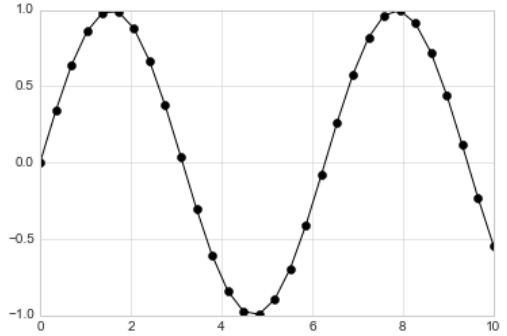
```
x = np.linspace(0, 10, 30)
y = np.sin(x)
plt.plot(x, y, 'o', color='black’);
```



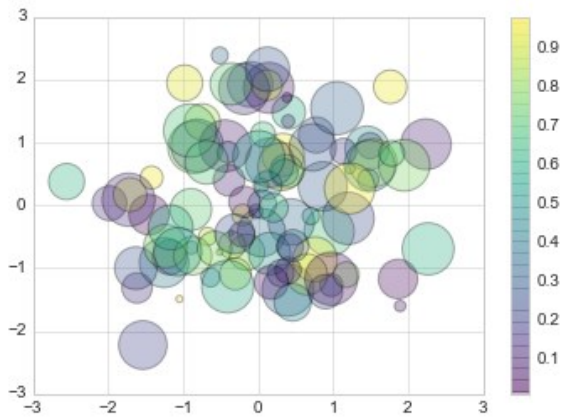
The third argument in the function call is a character that represents the type of symbol used for the plotting. Just as you can specify options such as '-' and '--' to control the line style, the marker style has its own set of short string codes. The full list of available symbols can be seen in the documentation of plt.plot, or in Matplotlib’s online documentation.

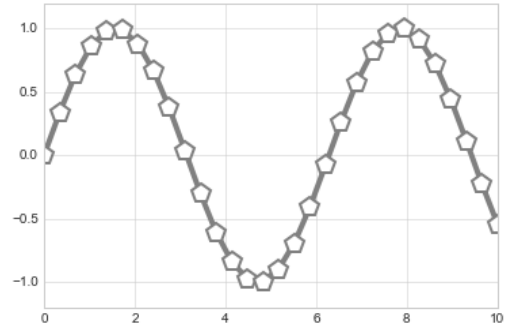
For even more possibilities, these character codes can be used together with line and
color codes to plot points along with a line connecting them.

```
plt.plot(x, y, '-ok'); # line (-), circle marker (o), black (k)
```




```
Cx = rng.randn(100)
y = rng.randn(100)
colors = rng.rand(100)
sizes = 1000 * rng.rand(100)
plt.scatter(x, y, c=colors, s=sizes, alpha=0.3,
cmap='viridis')
plt.colorbar(); # show color scaleustomizing line and point numbers
```





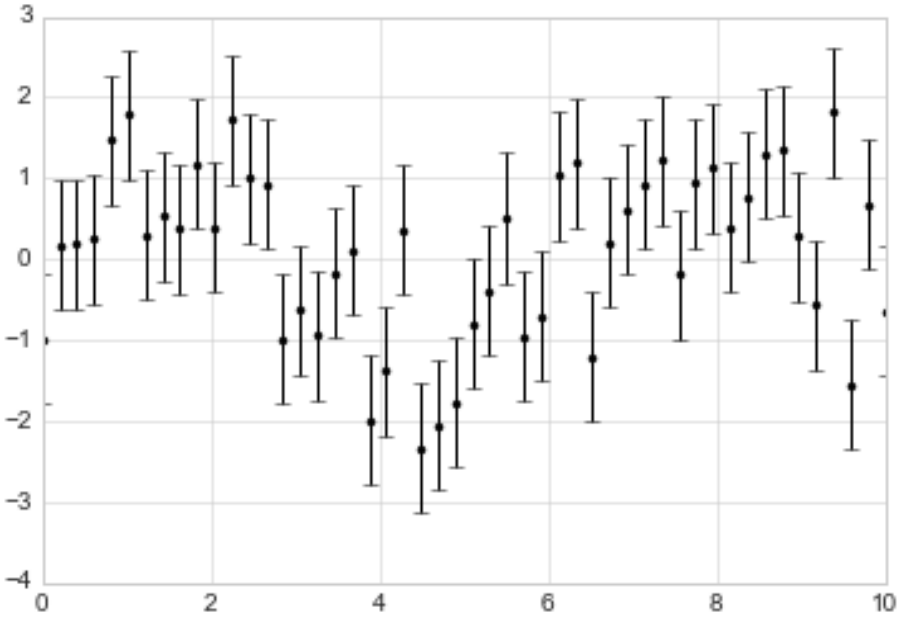
Customizing line and point numbers

Visualizing Errors

Basic Errorbars

A basic errorbar can be created with a single Matplotlib function call

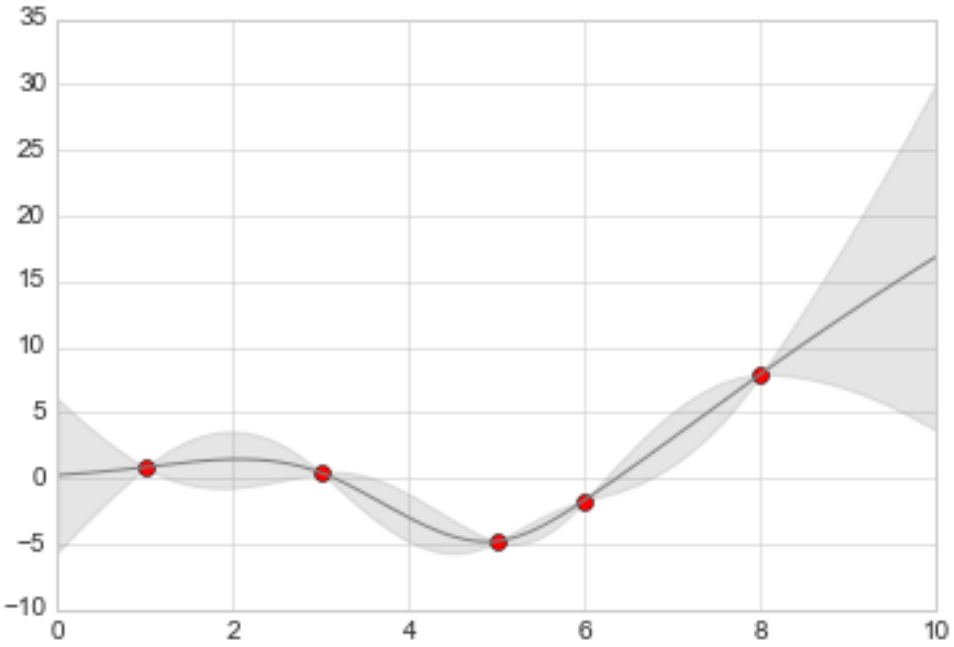
```
%matplotlib inline
import matplotlib.pyplot as plt
plt.style.use('seaborn-whitegrid')
import numpy as np
In[2]: x = np.linspace(0, 10, 50)
dy = 0.8
y = np.sin(x) + dy *
np.random.randn(50)
plt.errorbar(x, y, yerr=dy, fmt='.k');
```



Continuous Errors

We now have xfit, yfit, and dyfit, which sample the continuous fit to our data. We could pass these to the plt.errorbar function as above, but we don't really want to plot 1,000 points with 1,000 errorbars. Instead, we can use the plt.fill_between function with a light color to visualize this continuous error .

```
# Visualize the result
plt.plot(xdata, ydata, 'or')
plt.plot(xfit, yfit, '-', color='gray')
plt.fill_between(xfit, yfit - dyfit, yfit + dyfit,
color='gray', alpha=0.2)
plt.xlim(0, 10);
```



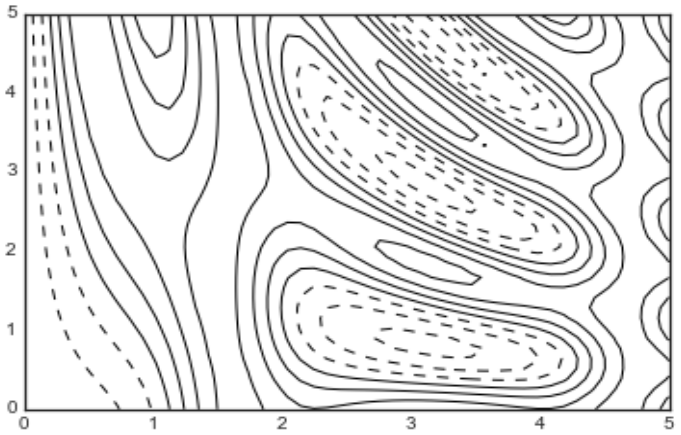
DENSITY AND CONTOUR PLOTS

Sometimes it is useful to display three-dimensional data in two dimensions using contours or color-coded regions. There are three Matplotlib functions that can be helpful for this task: plt.contour for contour plots, plt.contour for filled contour plots, and plt.imshow for showing images.

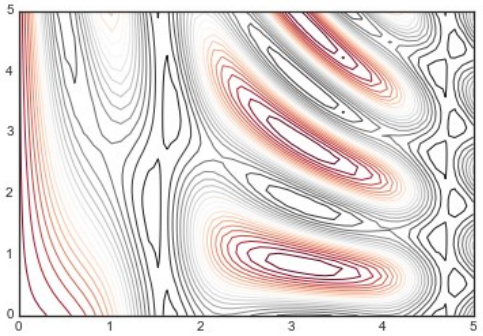
Visualizing a Three-Dimensional Function

There are three Matplotlib functions that can be helpful for this task: plt.contour for contour plots, plt.contourf for filled contour plots, and plt.imshow for showing images.

```
x = np.linspace(0, 5, 50)
y = np.linspace(0, 5, 40)
X, Y = np.meshgrid(x, y)
Z = f(X, Y)
plt.contour(X, Y, Z, colors='black');
```

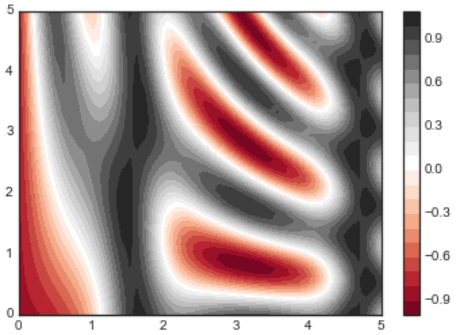


```
plt.contour(X, Y, Z, 20, cmap='RdGy');
```



RdGy (short for *Red-Gray*) colormap,

```
lt.contourf(X, Y, Z, 20, cmap='RdGy')
plt.colorbar();
```

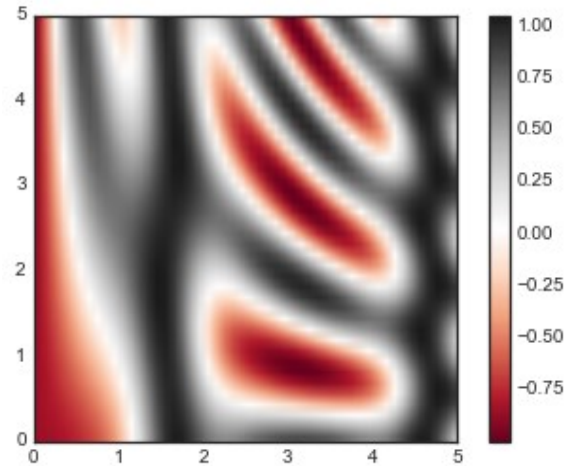


The colorbar makes it clear that the black regions are “peaks,” while the r
are “valleys.”

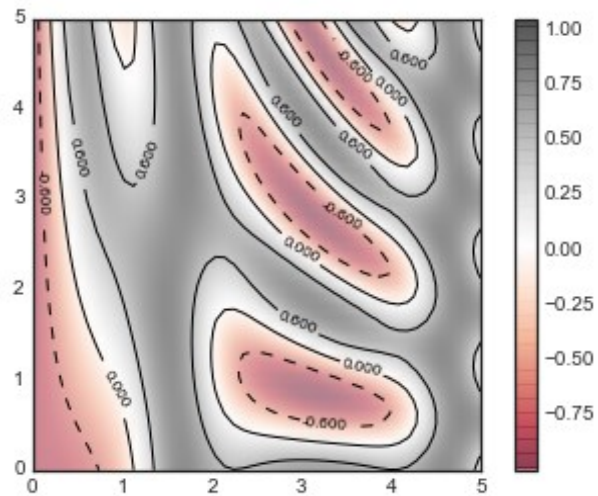
There are a few potential gotchas with imshow(), however:

- **plt.imshow()** doesn’t accept an *x* and *y* grid, so you must manually specify the *extent* [*xmin*, *xmax*, *ymin*, *ymax*] of the image on the plot.
- **plt.imshow()** by default follows the standard image array definition where the origin is in the upper left, not in the lower left as in most contour plots. This must be changed when showing gridded data.

Representing three-dimensional data as an image

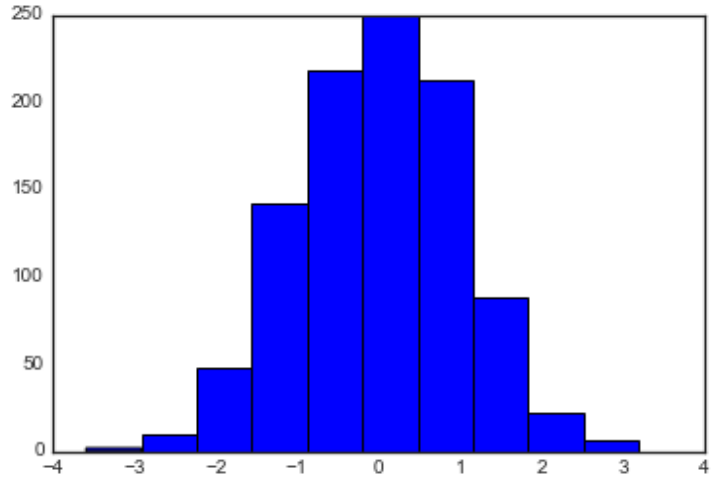


Labeled contours on top of an image



HISTOGRAMS, BINNINGS, AND DENSITY

```
%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);
```

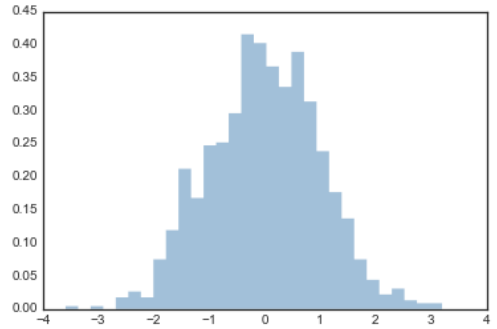


The hist() function has many options to tune both the calculation and the display;

```
plt.hist(data, bins=30, normed=True, alpha=0.5,
histtype='stepfilled', color='steelblue',
edgecolor='none');
```

The plt.hist docstring has more information on other customization options available. combination of histtype='stepfilled' along with some transparency

alpha to be very useful when comparing histograms of several distributions.



Two-Dimensional Histograms and Binnings

We create histograms in one dimension by dividing the number line into bins,

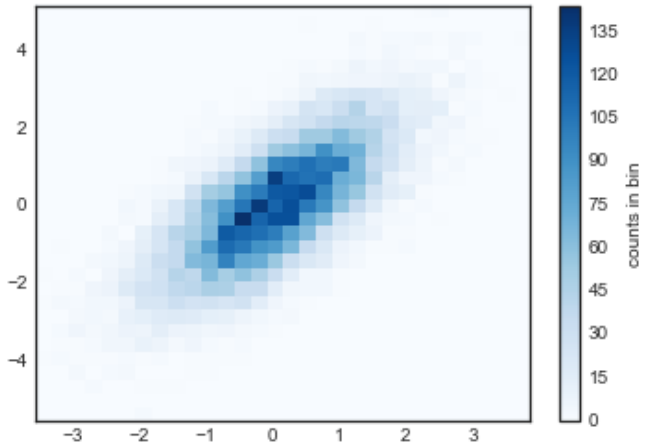
we can also create histograms in two dimensions by dividing points among two dimensional

bins. Defining some data—an x and y array drawn from a multivariate Gaussian distribution:

```
mean = [0, 0]
cov = [[1, 1], [1, 2]]
x, y = np.random.multivariate_normal(mean, cov, 10000).T
```

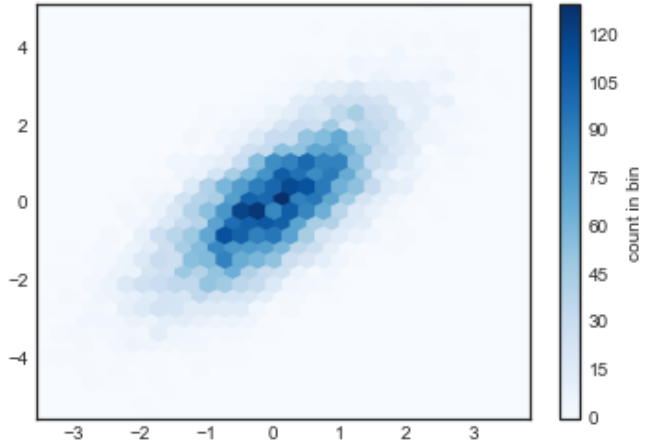
plt.hist2d: Two-dimensional histogram

```
plt.hist2d(x, y, bins=30, cmap='Blues')
cb = plt.colorbar()
cb.set_label('counts in bin’)
```



plt.hexbin: Hexagonal binnings

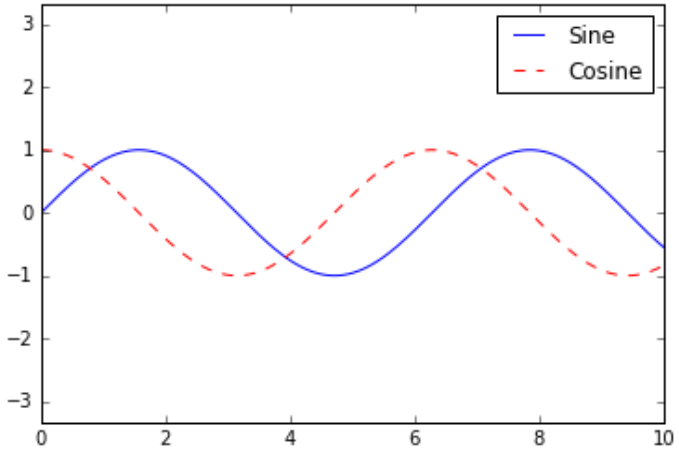
```
plt.hexbin(x, y, gridsize=30, cmap='Blues')
cb = plt.colorbar(label='count in bin')
```



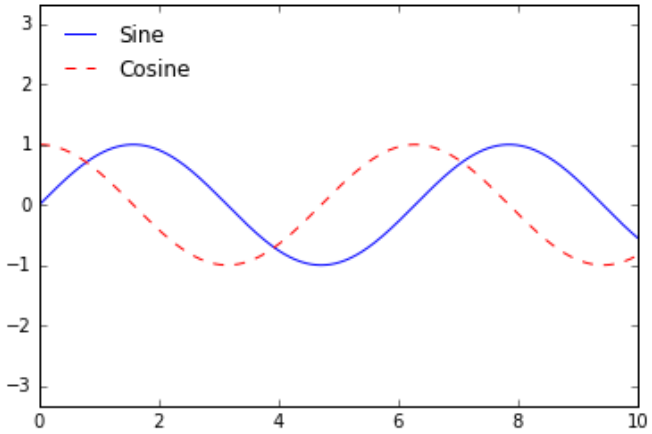
CUSTOMIZING PLOT LEGENDS

The simplest legend can be created with the plt.legend() command, which automatically

creates a legend for any labeled plot elements.

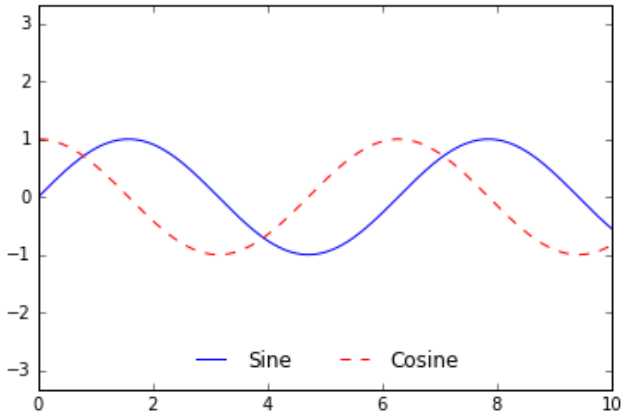



```
%matplotlib inline
import numpy as np
 x = np.linspace(0, 10, 1000)
fig, ax = plt.subplots()
ax.plot(x, np.sin(x), '-b', label='Sine')
ax.plot(x, np.cos(x), '--r', label='Cosine')
ax.axis('equal')
leg = ax.legend();
```




```
 ax.legend(loc='upper left', frameon=False)
```

A two-column plot legend

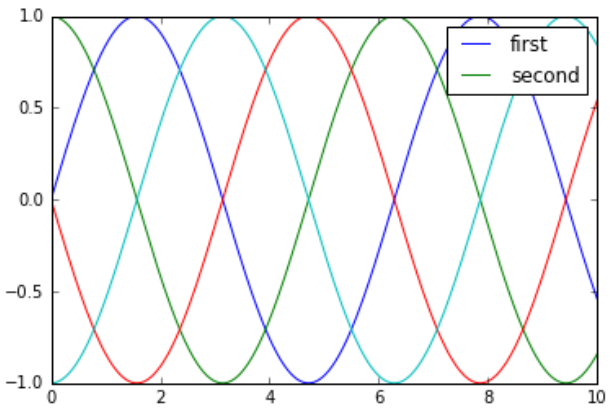


Choosing Elements for the Legend

```
y = np.sin(x[:, np.newaxis] + np.pi * np.arange(0, 2, 0.5))
lines = plt.plot(x, y)
plt.legend(lines[:2], ['first', 'second’]);
```

Alternative method:

```
plt.plot(x, y[:, 0], label='first')
plt.plot(x, y[:, 1], label='second')
plt.plot(x, y[:, 2:])
plt.legend(framealpha=1, frameon=True);
```



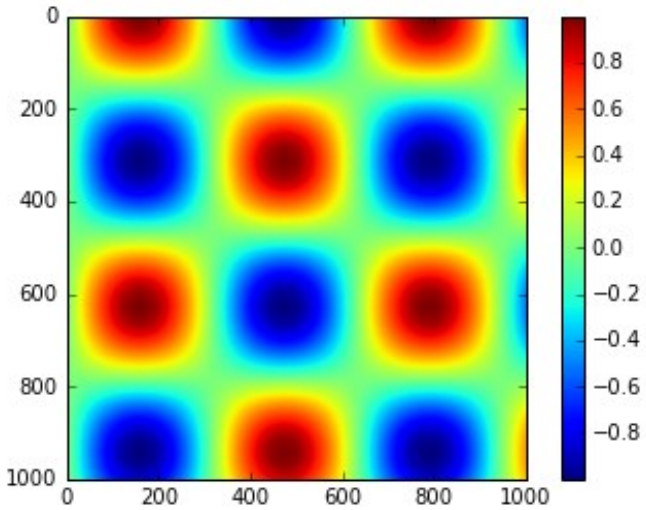
CUSTOMIZING COLORBARS

Plot legends identify discrete labels of discrete points. For continuous labels based on the color of points, lines, or regions, a labeled colorbar can be a great tool.

```
import matplotlib.pyplot as plt
plt.style.use('classic')
 %matplotlib inline
import numpy as np
```

The simplest colorbar can be created with the plt.colorbar function.

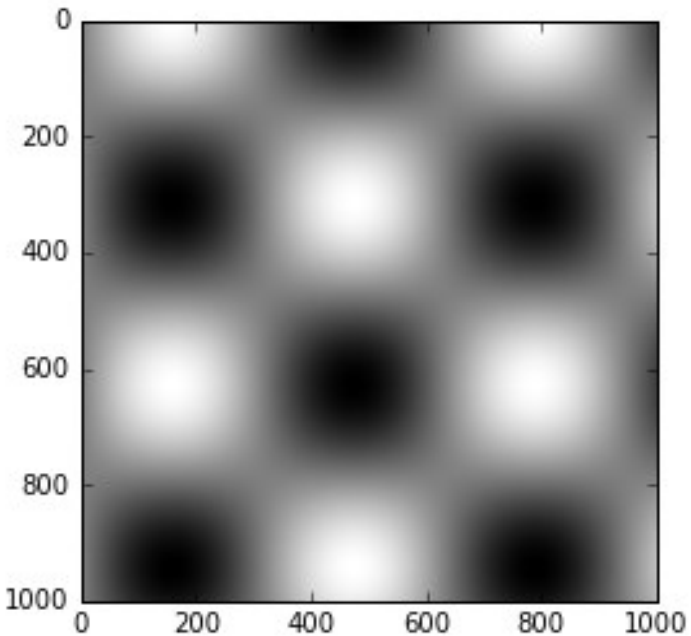
```
x = np.linspace(0, 10, 1000)
I = np.sin(x) * np.cos(x[:, np.newaxis])
plt.imshow(I)
plt.colorbar();
```



Customizing Colorbars

We can specify the colormap using the cmap argument to the plotting function that is creating the visualization . the available colormaps are in the plt.cm namespace; using IPython's tab completion feature will give you a full list of built-in possibilities:

plt.cm.<TAB>



Choosing the colormap

A full treatment of color choice within visualization

Matplotlib's online documentation also has an interesting discussion of colormap choice.

Broadly, you should be aware of three different categories of colormaps:

Sequential colormaps

These consist of one continuous sequence of colors (e.g., binary or viridis).

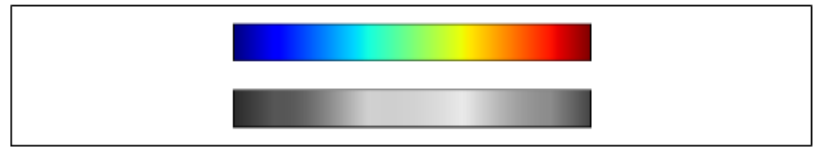
Divergent colormaps

These usually contain two distinct colors, which show positive and negative deviations from a mean (e.g., RdBu or PuOr).

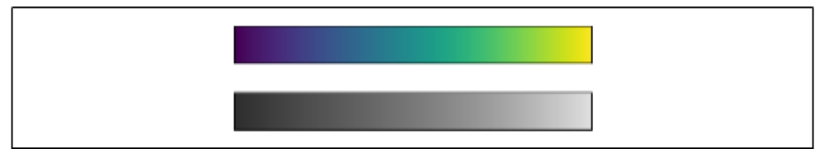
Qualitative colormaps

These mix colors with no particular sequence (e.g., rainbow or jet).

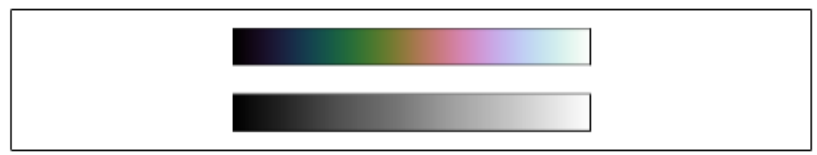
The jet colormap and its uneven luminance scale



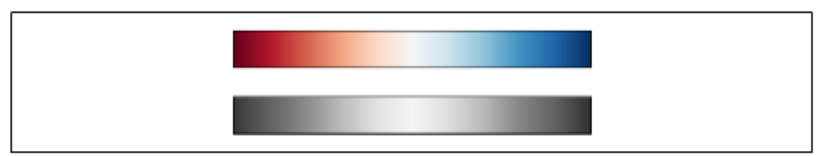
The viridis colormap and its even luminance scale



The cubehelix colormap and its luminance



The RdBu (Red-Blue) colormap and its luminance

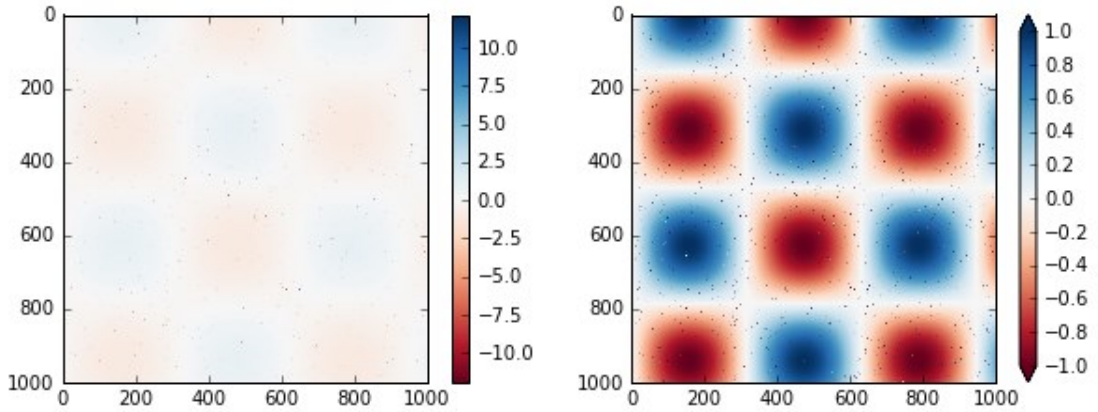


Color limits and extensions

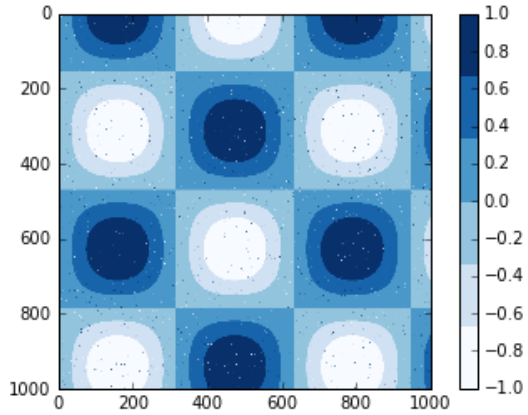
Matplotlib allows for a large range of colorbar customization. The colorbar itself is

simply an instance of plt.Axes, so all of the axes and tick formatting tricks . The colorbar has some interesting flexibility.

Specifying colormap extensions

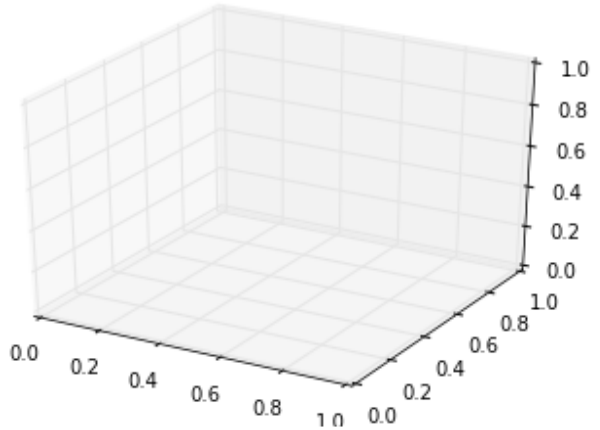


Discrete colorbars



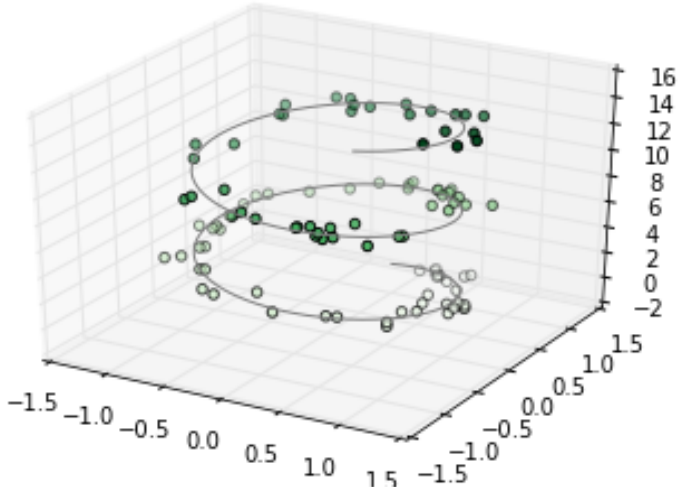
THREE-DIMENSIONAL PLOTTING IN MATPLOTLIB

An empty three-dimensional axes



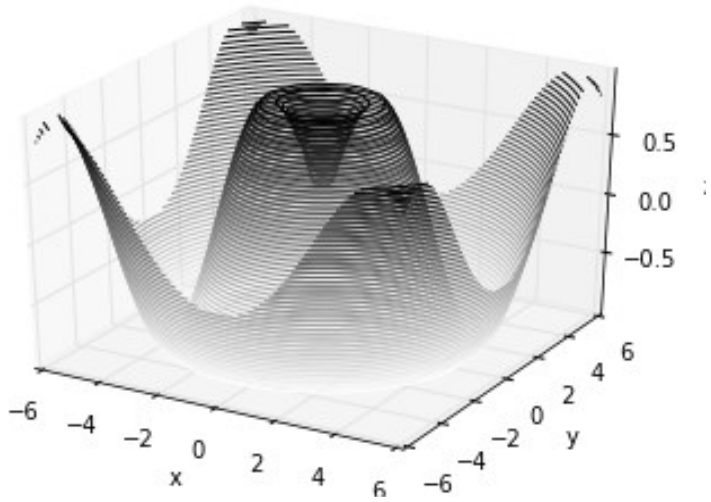
Three-Dimensional Points and Lines

```
ax = plt.axes(projection='3d')
# Data for a three-dimensional line
zline = np.linspace(0, 15, 1000)
xline = np.sin(zline)
yline = np.cos(zline)
ax.plot3D(xline, yline, zline, 'gray')
# Data for three-dimensional scattered points
zdata = 15 * np.random.random(100)
xdata = np.sin(zdata) + 0.1 * np.random.randn(100)
ydata = np.cos(zdata) + 0.1 * np.random.randn(100)
ax.scatter3D(xdata, ydata, zdata, c=zdata, cmap='Greens');
```



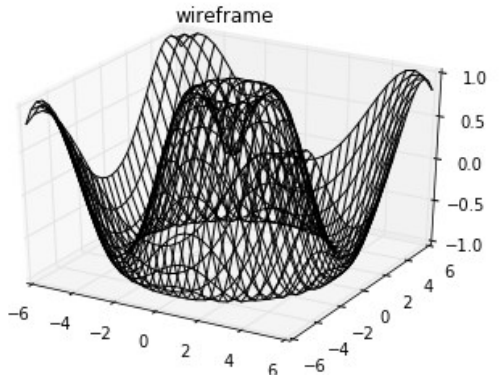
THREE-DIMENSIONAL CONTOUR PLOTS

```
def f(x, y):
return np.sin(np.sqrt(x ** 2 + y ** 2))
x = np.linspace(-6, 6, 30)
y = np.linspace(-6, 6, 30)
X, Y = np.meshgrid(x, y)
Z = f(X, Y)
In[6]: fig = plt.figure()
ax = plt.axes(projection='3d')
ax.contour3D(X, Y, Z, 50, cmap='binary')
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel('z');
```



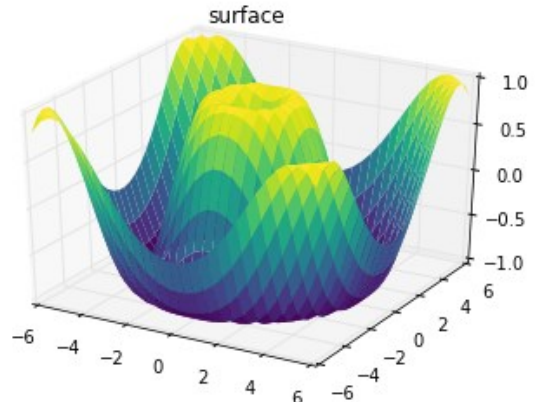
Wireframes and Surface Plots

```
fig = plt.figure()
ax = plt.axes(projection='3d')
ax.plot_wireframe(X, Y, Z, color='black')
ax.set_title('wireframe’);
```



A three-dimensional surface plot

```
ax = plt.axes(projection='3d')
ax.plot_surface(X, Y, Z, rstride=1, cstride=1,
cmap='viridis', edgecolor='none')
ax.set_title('surface');
```



A polar surface plot

```
r = np.linspace(0, 6, 20)
theta = np.linspace(-0.9 * np.pi, 0.8 * np.pi, 40)
r, theta = np.meshgrid(r, theta)
X = r * np.sin(theta)
Y = r * np.cos(theta)
Z = f(X, Y)
ax = plt.axes(projection='3d')
ax.plot_surface(X, Y, Z, rstride=1, cstride=1,
cmap='viridis', edgecolor='none');
```