7 - Matplotlib

September 3, 2020

1 Programming for Data Science and Artificial Intelligence

1.1 7. Matplotlib

1.1.1 Readings:

- [VANDER] Ch4
- https://matplotlib.org/3.2.2/contents.html

Matplotlib is a data visualization library built on NumPy arrays. It was conceived by John Hunter in 2002, originally as a patch to IPython for enabling interactive MATLAB-style plotting via gnuplot from the IPython command line. One of Matplotlib's most important features is its ability to play well with many operating systems and graphics backends. Matplotlib supports dozens of backends and output types, which means you can count on it to work regardless of which operating system you are using or which output format you wish.

Recent Matplotlib versions make it relatively easy to set new global plotting styles, and people have been developing new packages that build on its powerful internals to drive Matplotlib via cleaner, more modern APIs—for example, Seaborn, ggpy, HoloViews, Altair, and even Pandas itself can be used as wrappers around Matplotlib's API. Even with wrappers like these, it is still often useful to dive into Matplotlib's syntax to adjust the final plot output.

1.1.2 Importing Matplotlib

[1]:
```
import matplotlib

matplotlib.use('Agg')

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

1.1.3 Setting Styles

[2]:
```
#listing all possible styles
plt.style.available
```

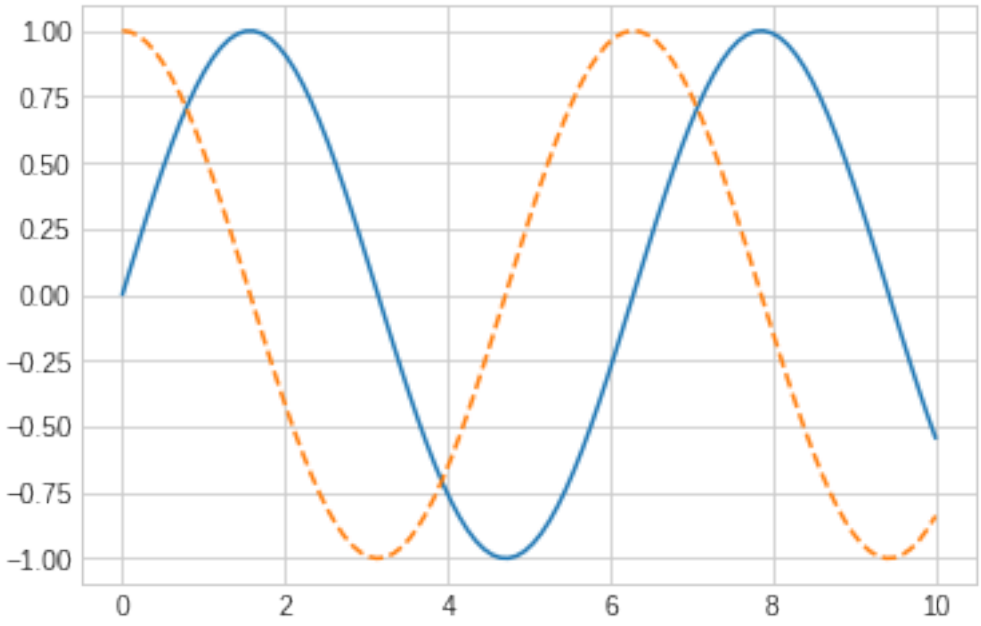
[2]:
```
['Solarize_Light2',
 '_classic_test_patch',
 'bmh',
 'classic',
 'dark_background',
 'fast',
 'fivethirtyeight',
 'ggplot',
 'grayscale',
 'seaborn',
 'seaborn-bright',
 'seaborn-colorblind',
 'seaborn-dark',
 'seaborn-dark-palette',
 'seaborn-darkgrid',
 'seaborn-deep',
 'seaborn-muted',
 'seaborn-notebook',
 'seaborn-paper',
 'seaborn-pastel',
 'seaborn-poster',
 'seaborn-talk',
 'seaborn-ticks',
 'seaborn-white',
 'seaborn-whitegrid',
 'tableau-colorblind10']
```

[3]:
```
#Use plt.style.use() to choose appropriate aesthetic styles.
plt.style.use('seaborn-whitegrid')
```

[4]:
```
#%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
```

[5]:
```
#now the plot will embed a PNG image of the resulting graphic
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), '--')
```

[5]: [<matplotlib.lines.Line2D at 0x7f59fce344c0>]



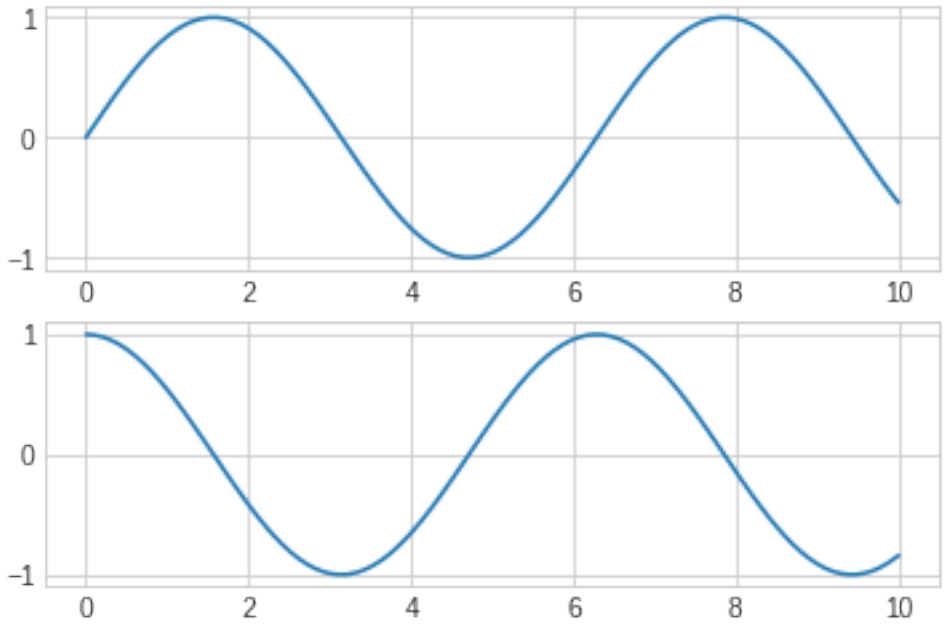
1.1.4 Two possible (confusing?) interface

MATLAB-style

[6]:
```
plt.figure()  #create a plot figure

#create the first of two panels and set current axis
plt.subplot(2, 1, 1)  #(row, columns, panel number)
plt.plot(x, np.sin(x))

#create the second panel
plt.subplot(2, 1, 2)
plt.plot(x, np.cos(x))

'''
- important to note that this api is stateful.  plt is a global variable
that keep track of all figure and axes.  One potential problem is that
it is difficult to edit the first figure once you come to the second figure
since the pointer has already pointed to the second plot
'''
```

[6]: '\n- important to note that this api is stateful. plt is a global variable\nthat keep track of all figure and axes. One potential problem is that\nit is difficult to edit the first figure once you come to the second

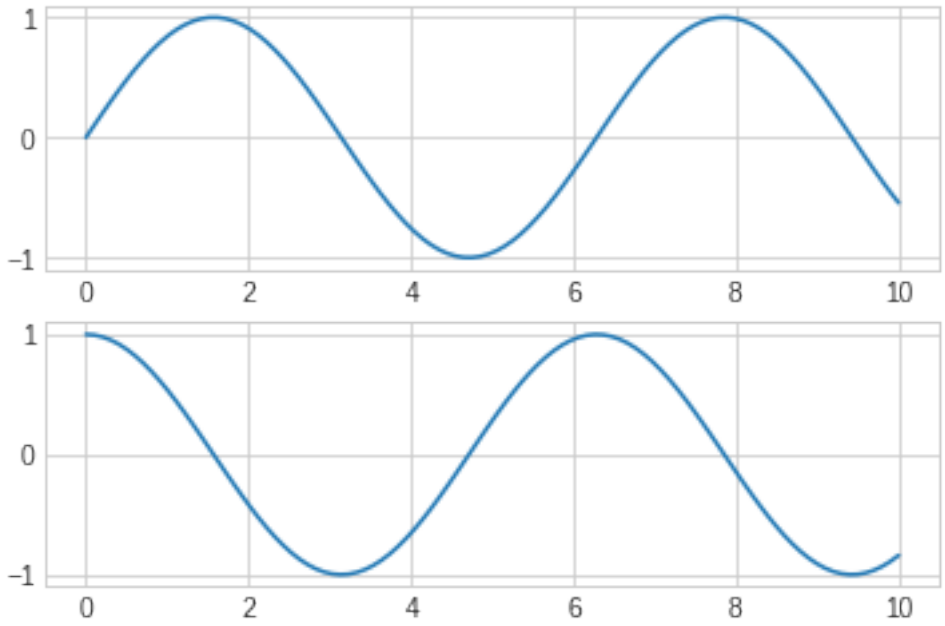
```
figure\nsince the pointer has already pointed to the second plot\n'
```



OO style

[7]:
```
# First create a grid of plots
# ax will be an array of two Axes objects
fig, ax = plt.subplots(2)  #notice the 's' there!!

# Call plot() method on the appropriate object
ax[0].plot(x, np.sin(x))
ax[1].plot(x, np.cos(x))

'''
In the OO style, it is working upon individual objects
'''
```

[7]:
```
'\nIn the OO style, it is working upon individual objects\n'
```



1.1.5 Simple Line Plots

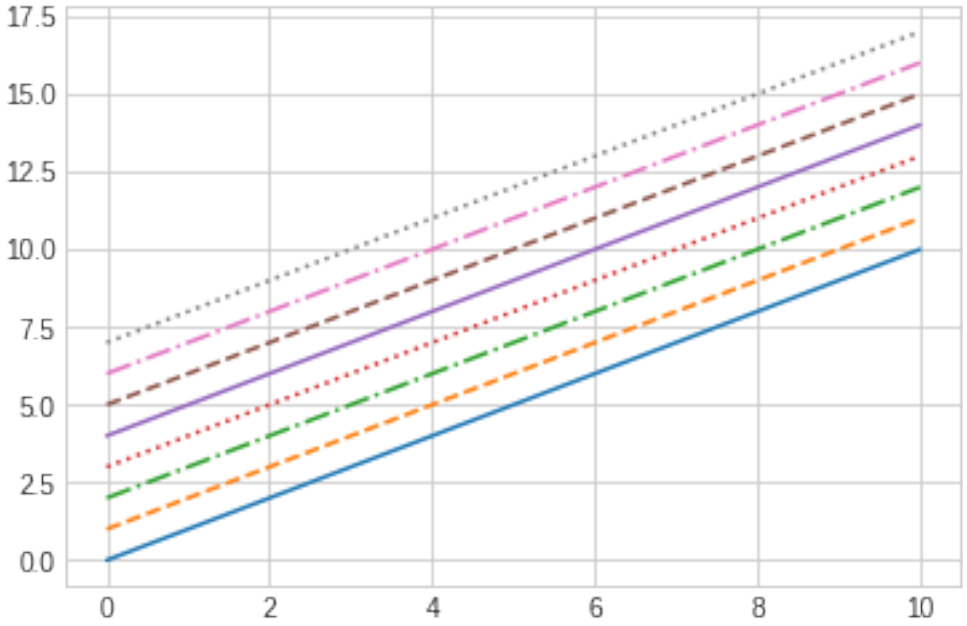
For all Matplotlib plots, we start by creating a figure and an axes. The figure is a single container that contains all the objects like axes, graphics, text, and labels. The axes represent the bounding box, e.g., ticks and labels.

[8]:
```
#object oriented way
import numpy as np

fig = plt.figure()
ax = plt.axes()  #get the instance of axes from plt

x = np.linspace(0, 10, 1000)
ax.plot(x, np.sin(x)) #x, y
ax.plot(x, np.cos(x))
```

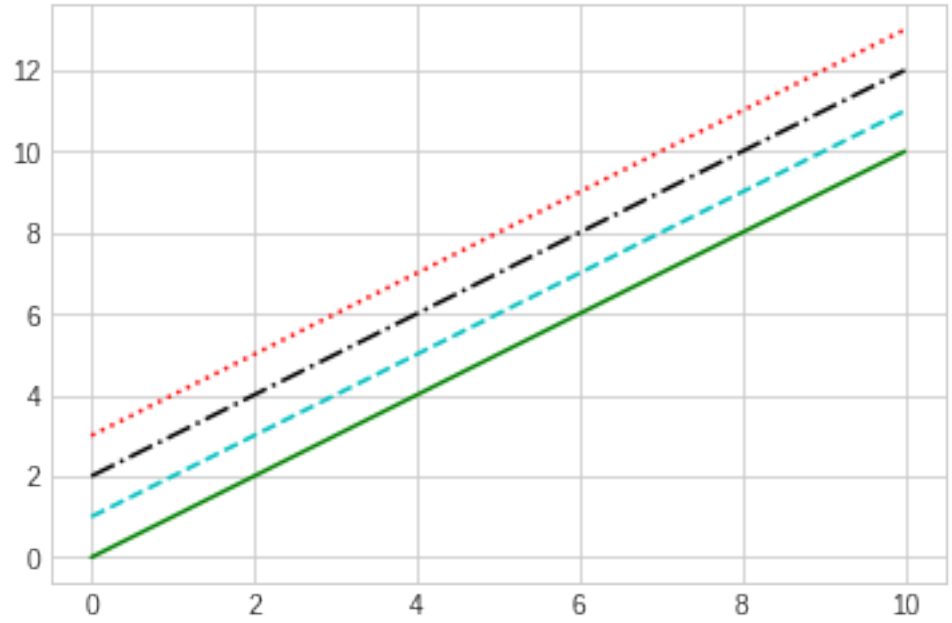
[8]:
```
[<matplotlib.lines.Line2D at 0x7f59fcc843d0>]
```

[9]:
```
#matlab way - actually simpler
plt.plot(x, np.sin(x))  #x, y
plt.plot(x, np.cos(x))

#many color options
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 to 1
plt.plot(x, np.sin(x - 5), color='chartreuse'); # all HTML color names supported
```

[10]:
```
#line options
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

#check help(plt.plot)  for all possible styles
```

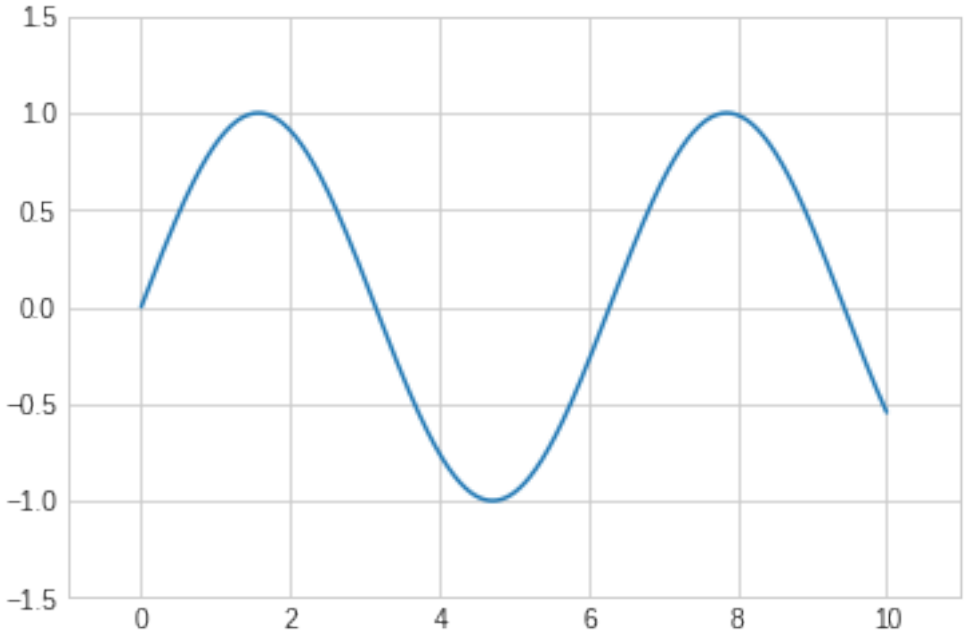


[11]:
```
#combine color and line options
plt.plot(x, x + 0, 'g-')  # solid green  #order does not matter
plt.plot(x, x + 1, '--c') # dashed cyan
plt.plot(x, x + 2, '-.k') # dashdot black
plt.plot(x, x + 3, ':r');  # dotted red
```

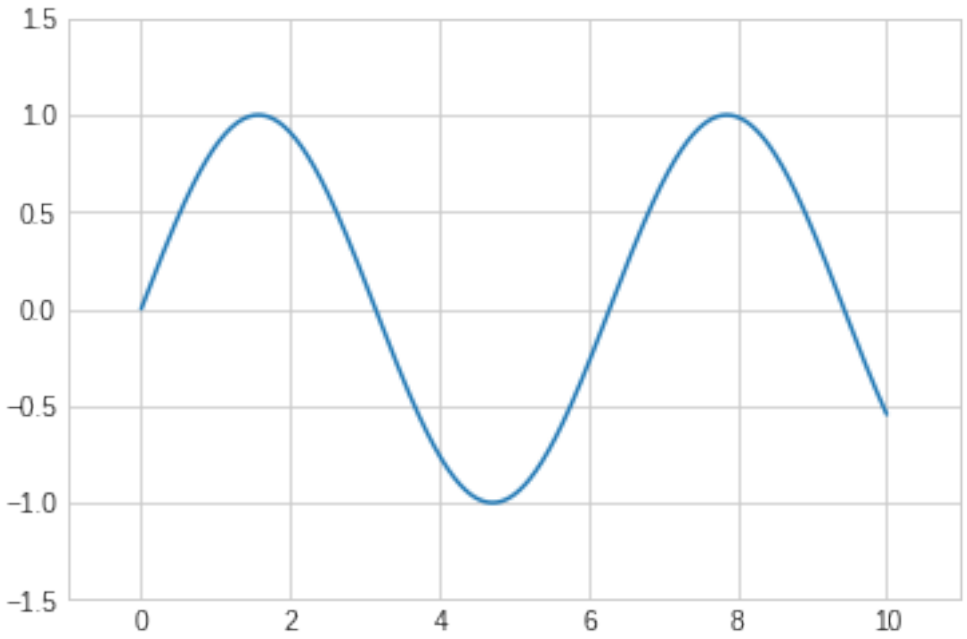


[12]:
```
#plot x y limits
plt.plot(x, np.sin(x))

plt.xlim(-1, 11)
plt.ylim(-1.5, 1.5) #you can also reverse!
```

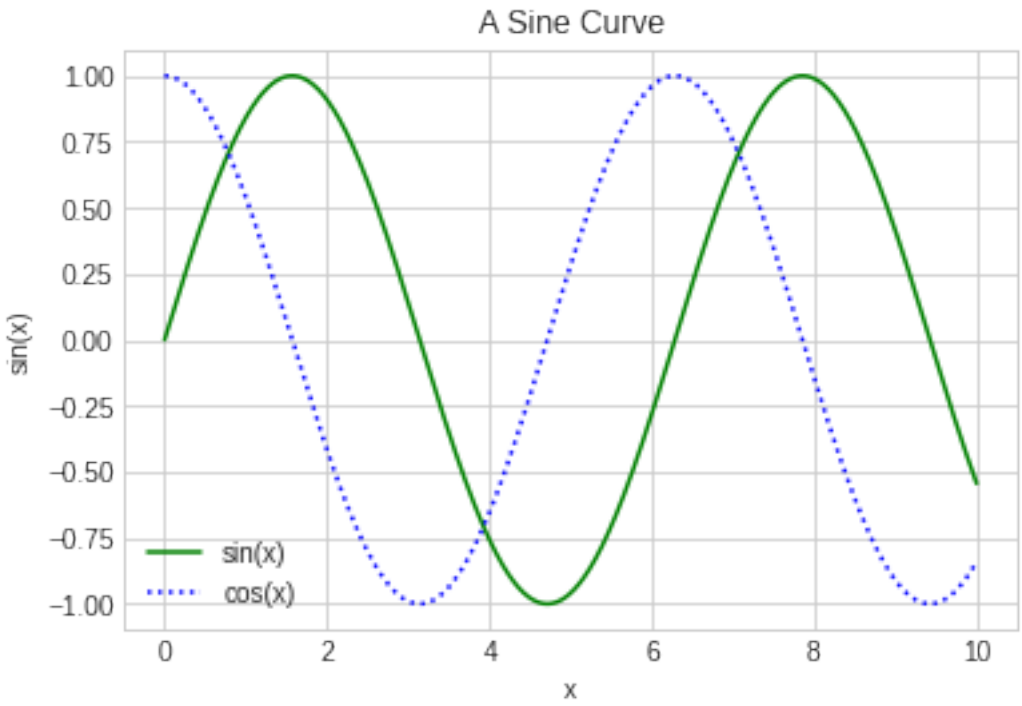
[12]: (-1.5, 1.5)



[13]:
```
#lazy to write two sentence, use plt.axis!
plt.plot(x, np.sin(x))
plt.axis([-1, 11, -1.5, 1.5]);  #xlim1, xlim2, ylim1, ylim2
# plt.axis('tight')  #other options  - 'equal'; help(plt.axis)
```



[14]:
```
#labeling
plt.plot(x, np.sin(x), '-g', label='sin(x)')
plt.plot(x, np.cos(x), ':b', label='cos(x)')
plt.title("A Sine Curve")
plt.legend()
plt.xlabel("x")
plt.ylabel("sin(x)");
```

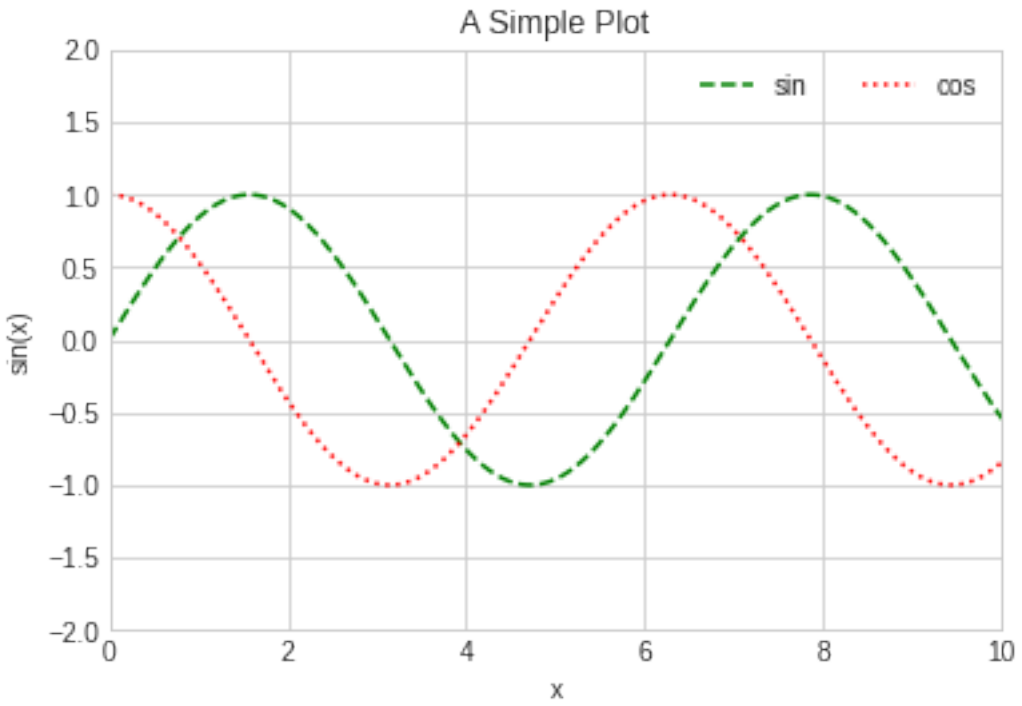


How about OO styles? Most plt functions translate directly to ax methods (such as plt.plot() → ax.plot(), plt.legend() → ax.legend(), etc.), this is not the case for all commands. In particular, functions to set limits, labels, and titles are slightly modified. For transitioning between MATLAB-style functions and object-oriented methods, make the following changes:

```
plt.xlabel() → ax.set_xlabel()
plt.ylabel() → ax.set_ylabel()
plt.xlim() → ax.set_xlim()
plt.ylim() → ax.set_ylim()
plt.title() → ax.set_title()
```

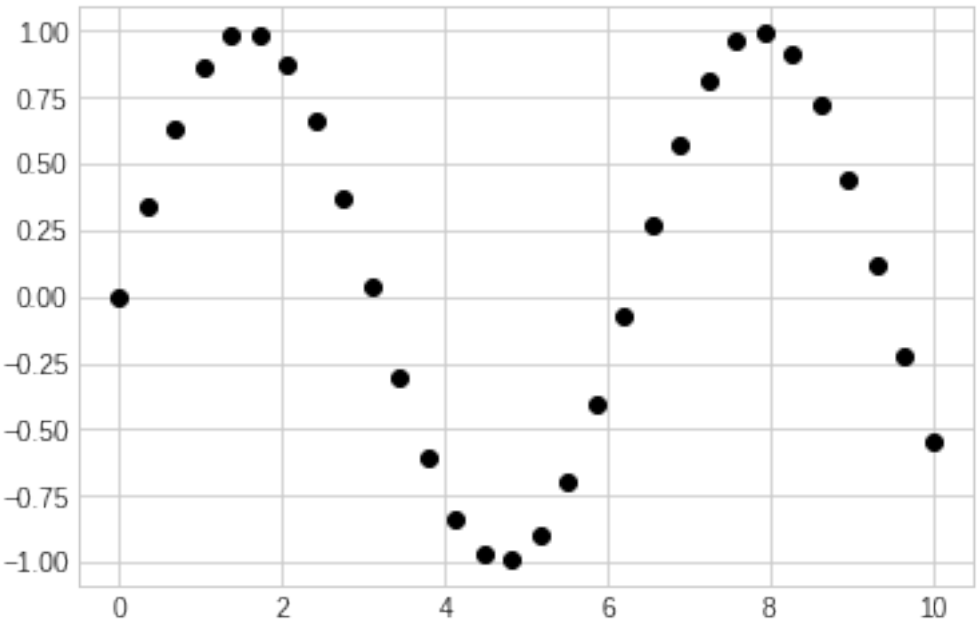
In the object-oriented interface to plotting, rather than calling these functions individually, it is more convenient to use the ax.set() method, to set everything at once:

[15]:
```
ax = plt.axes()
ax.plot(x, np.sin(x), '--g', label='sin')  #label for legend
ax.plot(x, np.cos(x), ':r', label='cos')
ax.legend(ncol = 2)
ax.set(xlim=(0, 10), ylim=(-2, 2),
       xlabel='x', ylabel='sin(x)',   #label for axis
       title='A Simple Plot');
```



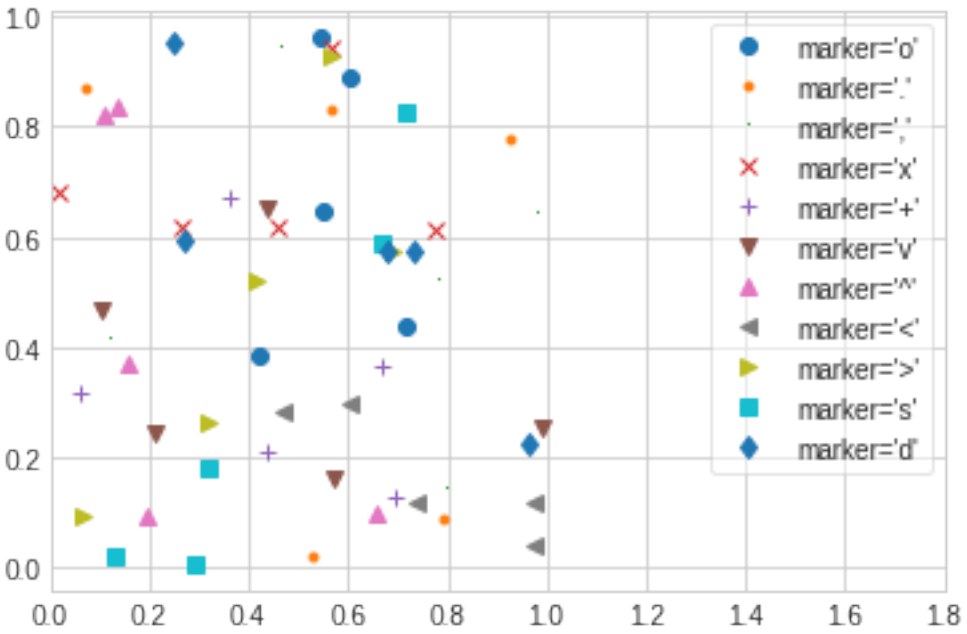
1.1.6 Scatter plots

[16]:
```
#we can simply use plt.plot but with different shapes to create scatter plot
x = np.linspace(0, 10, 30)
y = np.sin(x)

plt.plot(x, y, 'o', color='black');
```

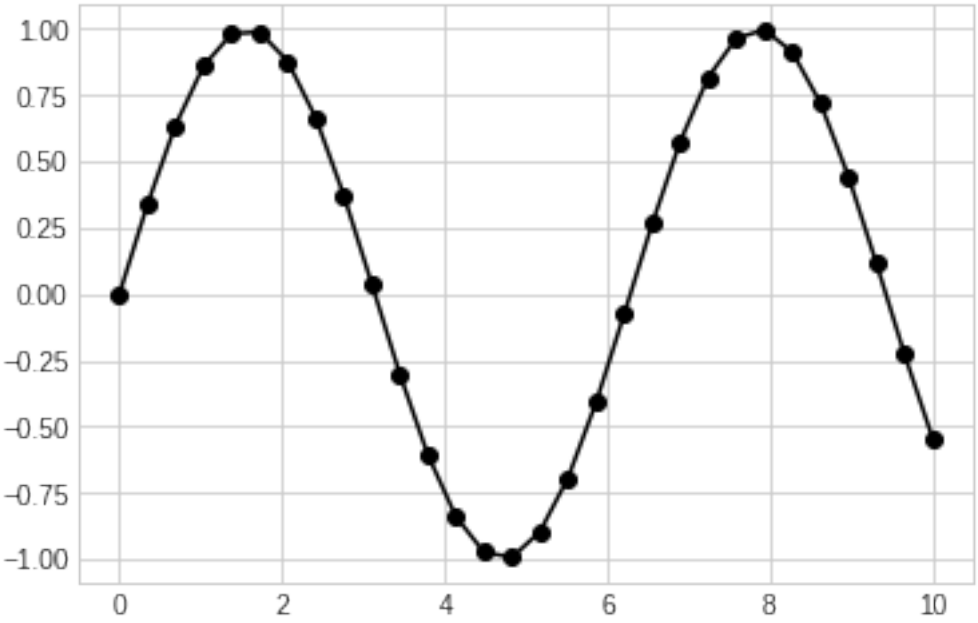


[17]:
```
#many possible markers!
rng = np.random.RandomState(0)
for marker in ['o', '.', ',', 'x', '+', 'v', '^', '<', '>', 's', 'd']:
    plt.plot(rng.rand(5), rng.rand(5), marker,
             label="marker='{0}'".format(marker))
plt.legend(loc='upper right', frameon=True,
          fancybox=True, framealpha=0.5)  #can accept parameters like location,␣
 ↪whethers has border, round edges
plt.xlim(0, 1.8)
```

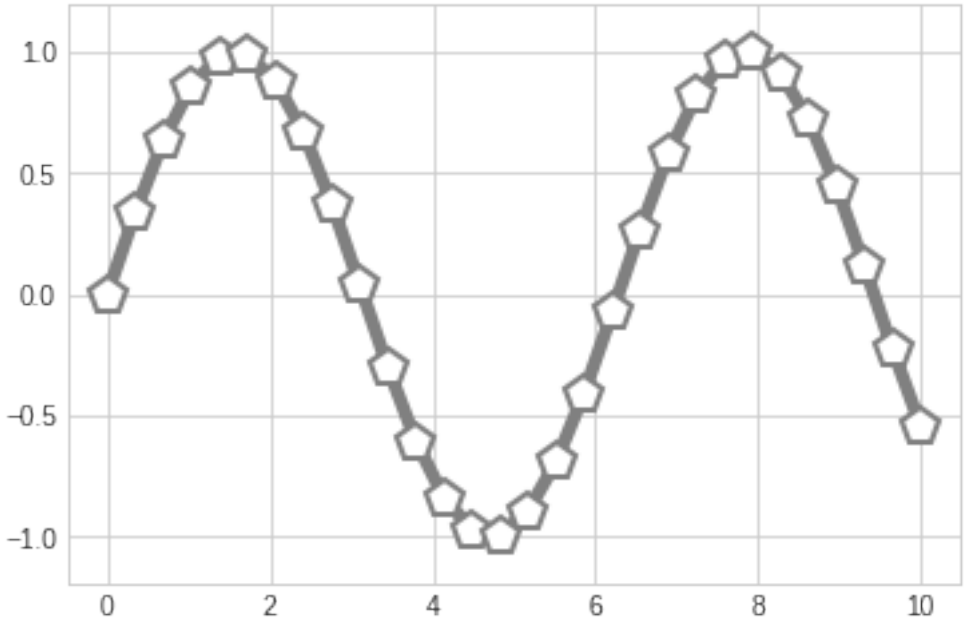
[17]: (0.0, 1.8)



[18]:
```
#combine colors, line styles, and marker styles
plt.plot(x, y, '-ok');  #k = black, o = dots, - = line
```

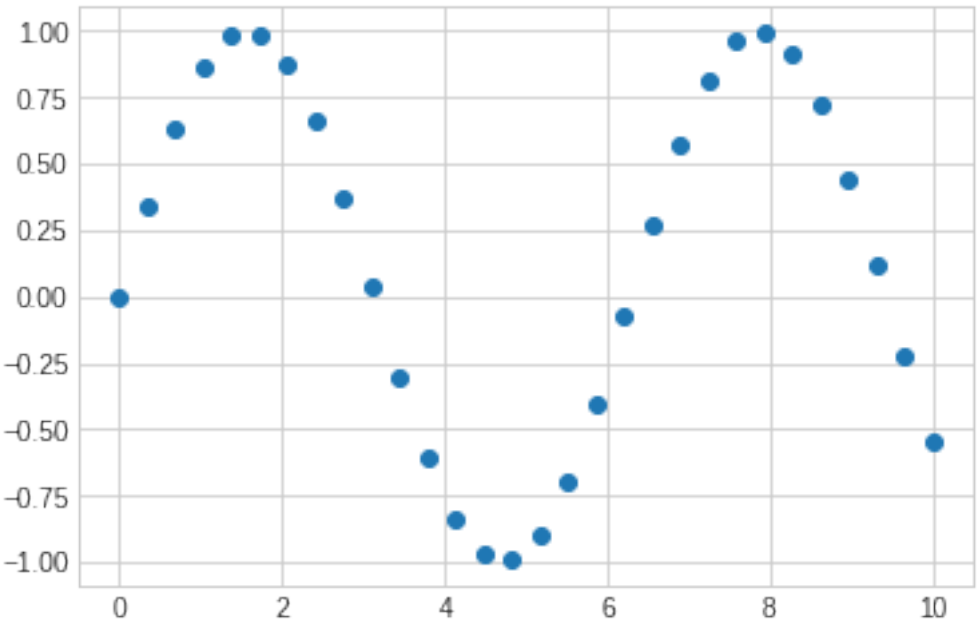


[19]:
```
#additional arguments like markersize, markerfacecolor, edgecolor
plt.plot(x, y, '-p', color='gray',    #p = pentagon
         markersize=15, linewidth=5,
         markerfacecolor='white',
         markeredgecolor='gray',
         markeredgewidth=2)
plt.ylim(-1.2, 1.2);
```



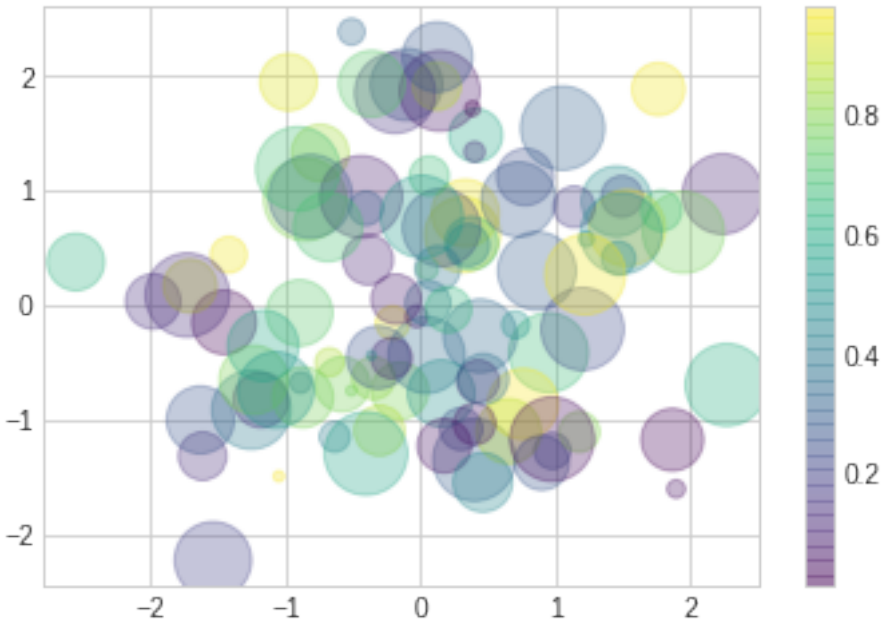
[20]:
```
##python guys also decided to create a wrapper for easier use of scatterplot
##providing plt.scatter
plt.scatter(x, y, marker='o')
```

[20]:
```
<matplotlib.collections.PathCollection at 0x7f59fcd9be20>
```



[21]:
```
## The primary difference of plt.scatter from plt.plot is that it can
## be used to create scatter plots where the properties of each individual
## point (size, face color, edge color, etc.) can be individually
## controlled or mapped to data.
#plt can also accept x, and y in the form of (n, ), also s = (n, )
rng = np.random.RandomState(0)
x = rng.randn(100)
x.shape  #shape of (n, )
y = rng.randn(100)
colors = rng.rand(100)
sizes = 1000 * rng.rand(100)

# print(x, y, sizes)

plt.scatter(x, y, c=colors, s=sizes, alpha=0.3,
            cmap='viridis')
plt.colorbar()  # show color scale
```

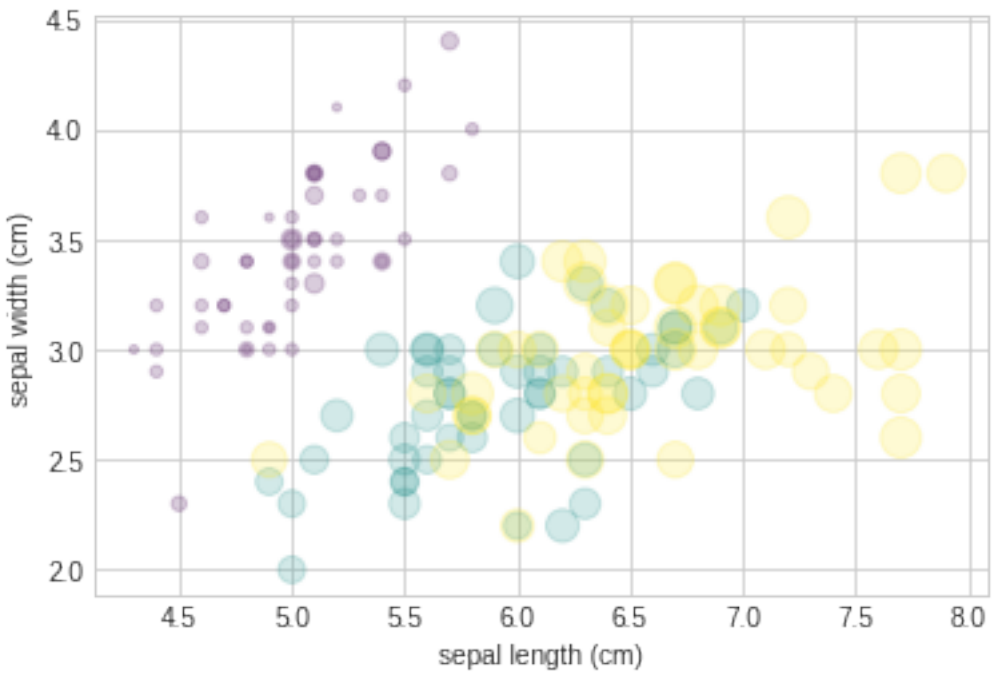
[21]:
```
<matplotlib.colorbar.Colorbar at 0x7f59fc2c05e0>
```



For example, we might use the Iris data from Scikit-Learn, where each sample is one of three types of flowers that has had the size of its petals and sepals carefully measured:

[22]:
```
from sklearn.datasets import load_iris
iris = load_iris()

#print(iris.data.shape)

features = iris.data.T

#print(features.shape)  #four features
#print(features[0].shape)

#sepal length and width for x1, x2
#size for petal width
#color for y
plt.scatter(features[0], features[1], alpha=0.2,
            s=100*features[3], c=iris.target, cmap='viridis')
plt.xlabel(iris.feature_names[0])
plt.ylabel(iris.feature_names[1])
```

[22]:
```
Text(0, 0.5, 'sepal width (cm)')
```

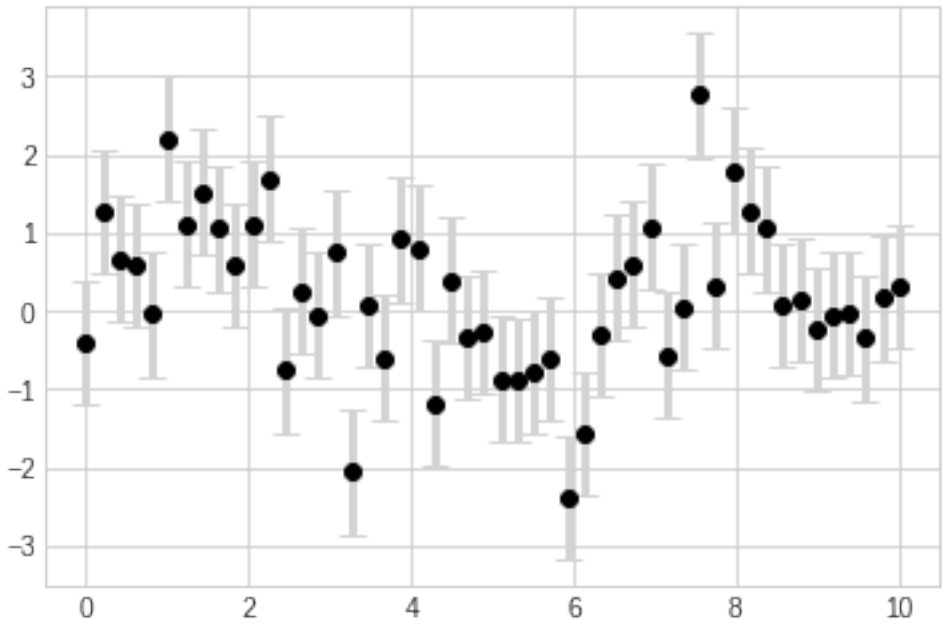


[23]:
```
##so which one you should use?  plt.plot is more efficient for big datasets
```

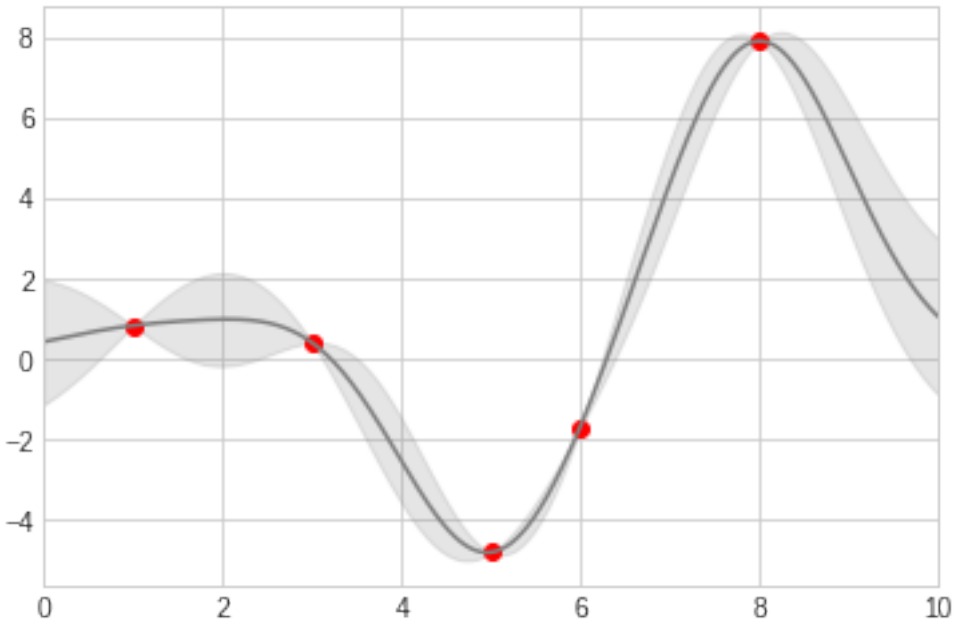
1.1.7 Errorbars

[24]:
```
#use plt.errorbar
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='ok', ecolor='lightgray',
             elinewidth=3, capsize=5)  #capsize is the bar on top and bot
```

[24]:
```
<ErrorbarContainer object of 3 artists>
```



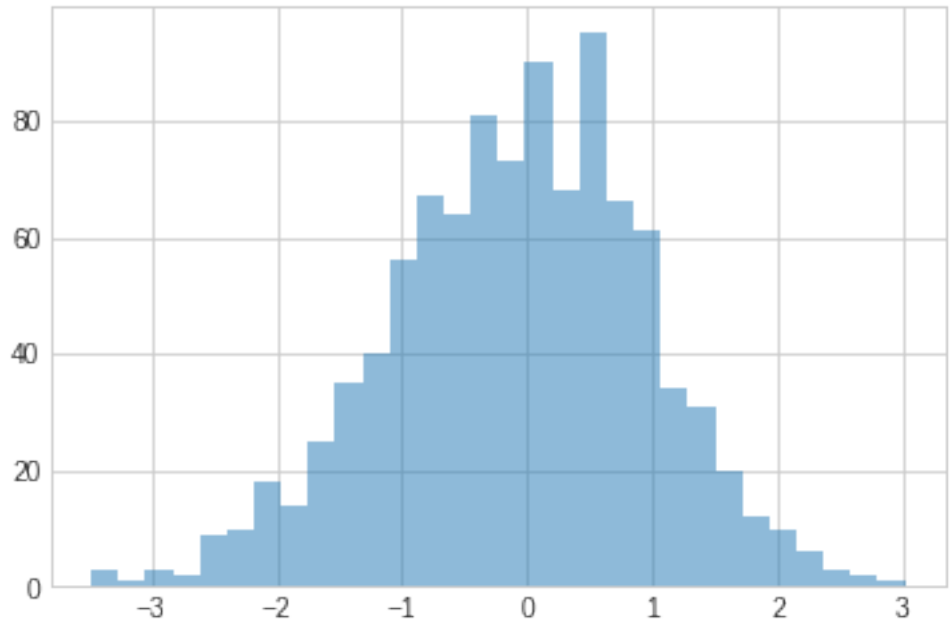
[25]:
```
### Continuous errors
#Let's say you plot a line, and you would like to know the areas of
#errors.  Use fill_between()

from sklearn.gaussian_process import GaussianProcessRegressor as GP

# define the model and draw some data
model = lambda x: x * np.sin(x)
xdata = np.array([1, 3, 5, 6, 8])
ydata = model(xdata)

# Compute the Gaussian process fit
gp = GP().fit(xdata[:, np.newaxis], ydata)

xfit = np.linspace(0, 10, 1000)
yfit, sigma = gp.predict(xfit[:, np.newaxis], return_std=True)
dyfit = 1.96 * sigma  # 95% confidence region

# Visualize the result
plt.plot(xdata, ydata, 'or')  #red circle
plt.plot(xfit, yfit, '-', color='gray')   #fitted by guassian

plt.fill_between(xfit, yfit - dyfit, yfit + dyfit,
                 color='gray', alpha=0.2)
plt.xlim(0, 10);
```



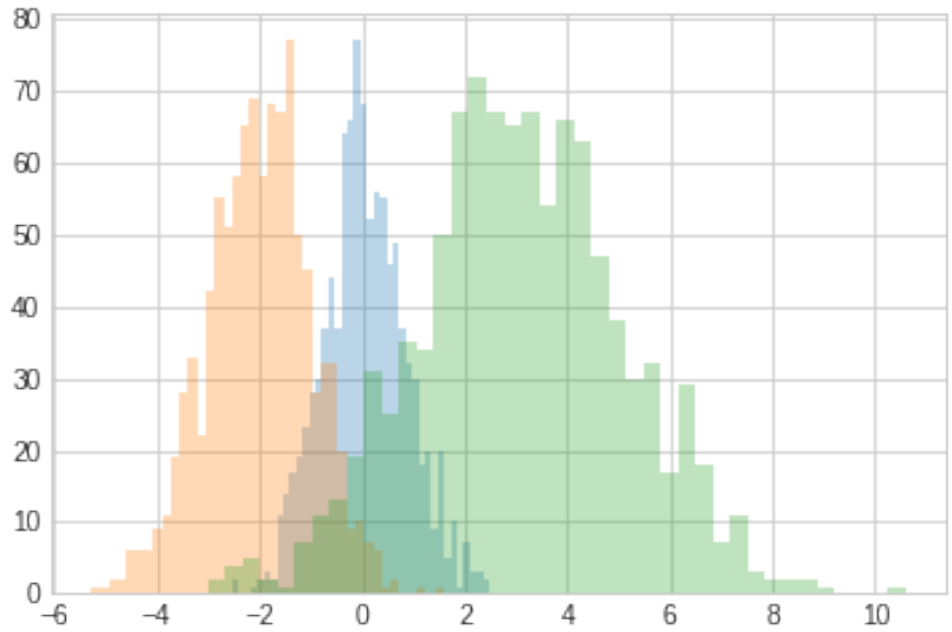
1.1.8 Histograms

[26]:
```
#A simple histogram can be a great first step in understanding a dataset.
data = np.random.normal(0, 1, 1000)  #1000 numbers between [0, 1)
plt.hist(data, bins=30, alpha=0.5)  #x should be in the shape of (n, )
```

[26]:
```
(array([ 3.,  1.,  3.,  2.,  9., 10., 18., 14., 25., 35., 40., 56., 67.,
        64., 81., 73., 90., 68., 95., 66., 61., 34., 31., 20., 12., 10.,
         6.,  3.,  2.,  1.]),
 array([-3.48157996, -3.26500049, -3.04842102, -2.83184156, -2.61526209,
        -2.39868262, -2.18210316, -1.96552369, -1.74894422, -1.53236475,
        -1.31578529, -1.09920582, -0.88262635, -0.66604689, -0.44946742,
        -0.23288795, -0.01630849,  0.20027098,  0.41685045,  0.63342991,
         0.85000938,  1.06658885,  1.28316831,  1.49974778,  1.71632725,
         1.93290672,  2.14948618,  2.36606565,  2.58264512,  2.79922458,
         3.01580405]),
 <a list of 30 Patch objects>)
```



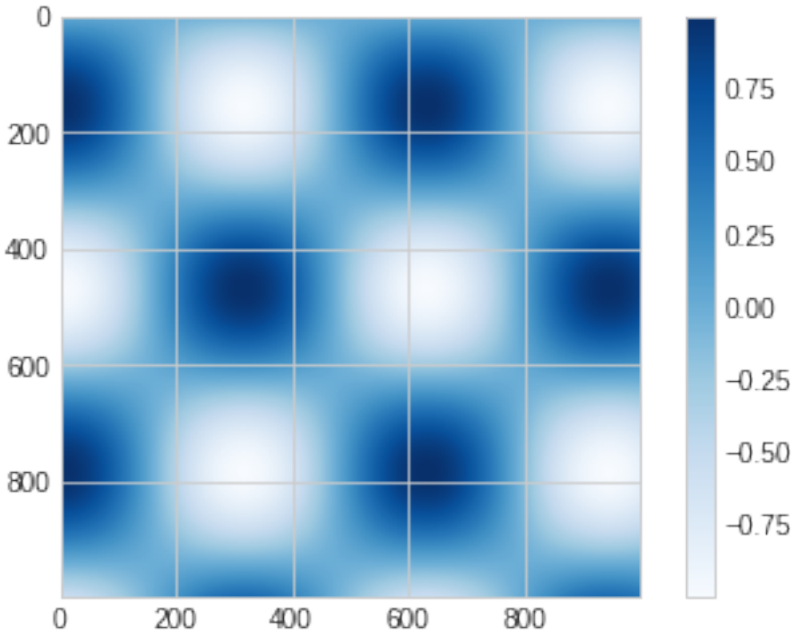
[27]:
```
#.hist also accepts a keyword arguments
x1 = np.random.normal(0, 0.8, 1000)
x2 = np.random.normal(-2, 1, 1000)
x3 = np.random.normal(3, 2, 1000)

kwargs = dict(alpha=0.3, bins=40)

plt.hist(x1, **kwargs)
plt.hist(x2, **kwargs)
plt.hist(x3, **kwargs);
```



1.1.9 Colorbars

[28]:
```
###colorbar created using plt.colorbar
x = np.linspace(0, 10, 1000)
I = np.sin(x[:, np.newaxis]) * np.cos(x)  #so that I is in shape of (M, N) ␣
 ↪where M and N can be any number

#a lot of cmap to choose
#try plt.cm.<TAB>
plt.imshow(I, cmap = 'Blues')
# plt.imshow(I, cmap=plt.cm.get_cmap('Blues', 6))  #for discrete color map

plt.colorbar()
```

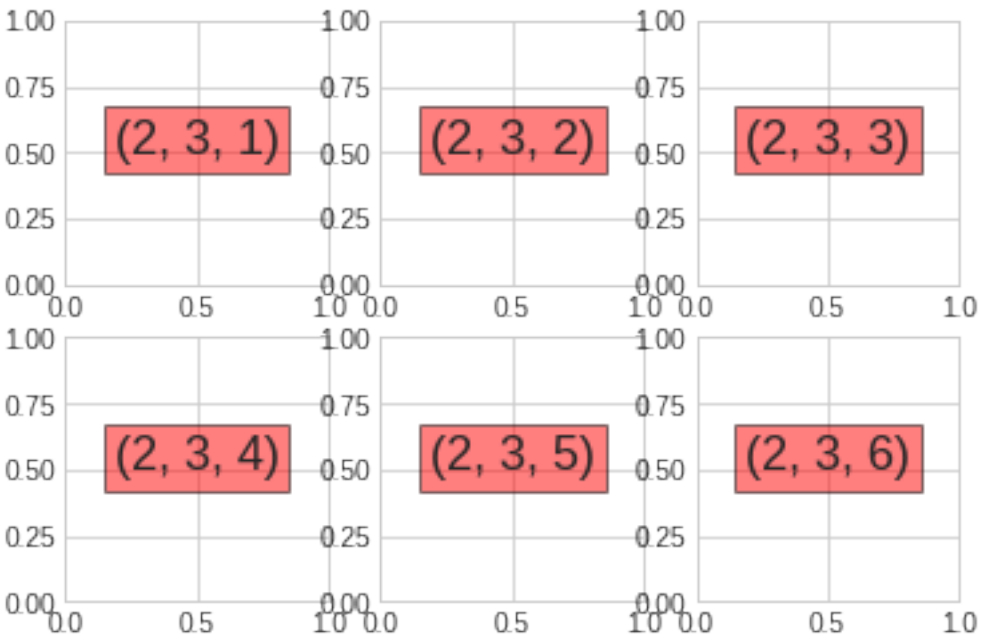
[28]:
```
<matplotlib.colorbar.Colorbar at 0x7f59f3d5f880>
```



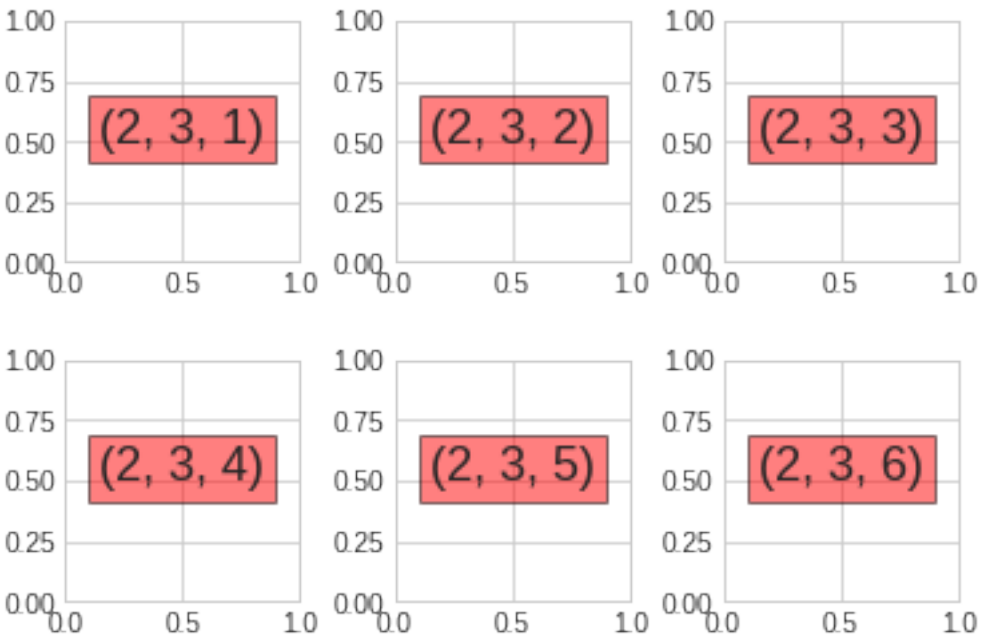
1.1.10 Multiple subplots

plt subplot takes three integer arguments—the number of rows, the number of columns, and the index of the plot to be created in this scheme, which runs from the upper left to the bottom right:

[29]:
```
#matlab style
for i in range(1, 7):
    plt.subplot(2, 3, i)
    #plt.text (x, y, string) (0, 0) lower left to (1, 1) upper right
    plt.text(0.5, 0.5, str((2, 3, i)),
             fontsize=18, horizontalalignment='center',
             bbox=dict(facecolor='red', alpha=0.5))
```

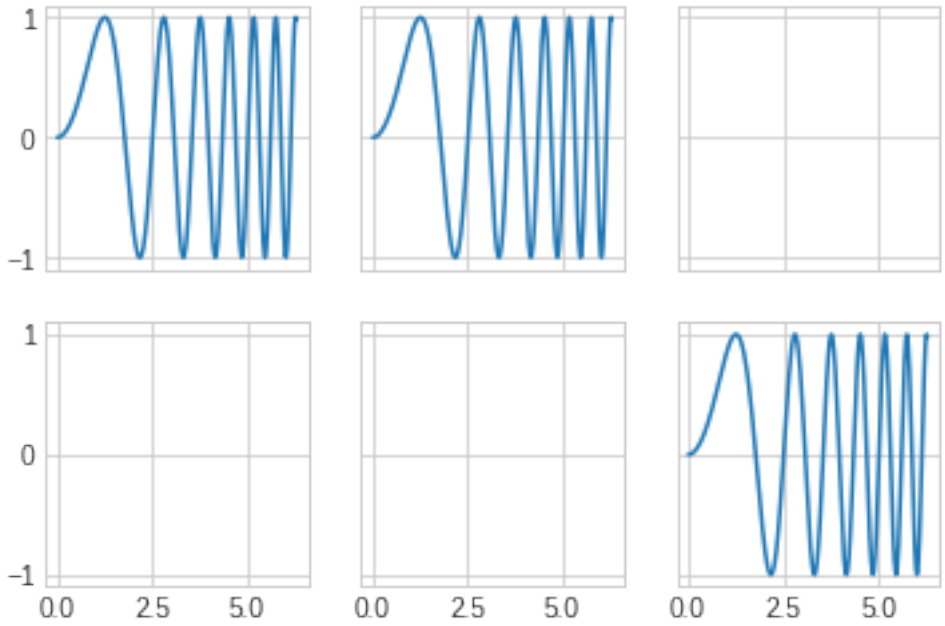


[30]:
```
#oo style
fig = plt.figure()
fig.subplots_adjust(hspace=0.4, wspace=0.4)
for i in range(1, 7):
    ax = fig.add_subplot(2, 3, i)
    ax.text(0.5, 0.5, str((2, 3, i)),
            fontsize=18, horizontalalignment='center',
            bbox=dict(facecolor='red', alpha=0.5))
```



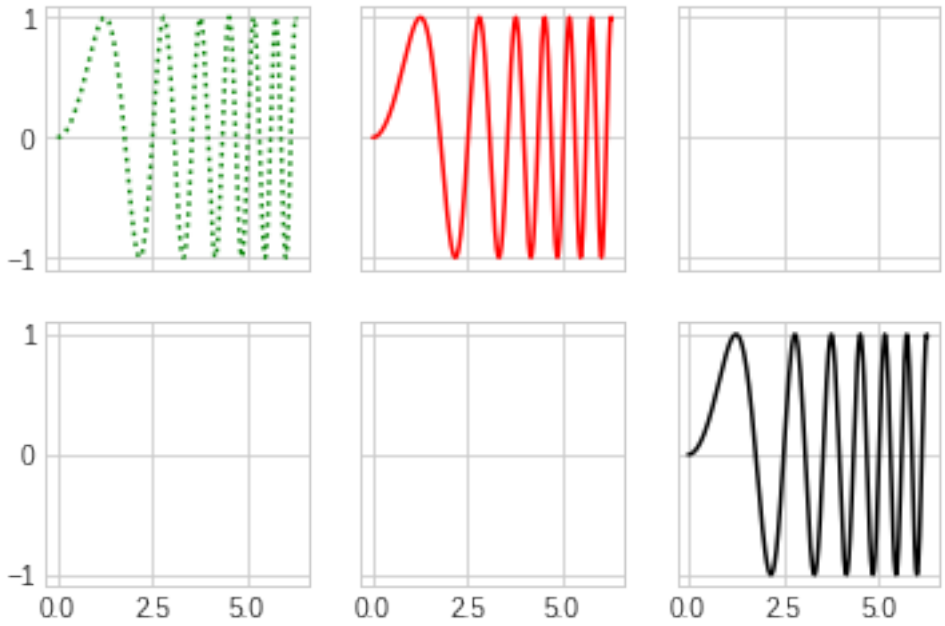
[31]:
```
#plt.subplot(s)  - Note the s at the end
#a more efficient way to plot many at the same time

# First create some toy data:
x = np.linspace(0, 2*np.pi, 400)
y = np.sin(x**2)

fig, ax = plt.subplots(2, 3, sharex='col', sharey='row')
ax[0, 0].plot(x, y)  #row cols
ax[0, 1].plot(x, y)
ax[1, 2].plot(x, y)
```

[31]: [<matplotlib.lines.Line2D at 0x7f59f3b305b0>]



[32]:
```
#use tuple way
#return ax1 for first row, and ax2 for second row
fig, (ax1, ax2) = plt.subplots(2, 3, sharex='col', sharey='row')
ax1[0].plot(x, y, ':g')
ax1[1].plot(x, y, '-r')
ax2[2].plot(x, y, '-k')
```

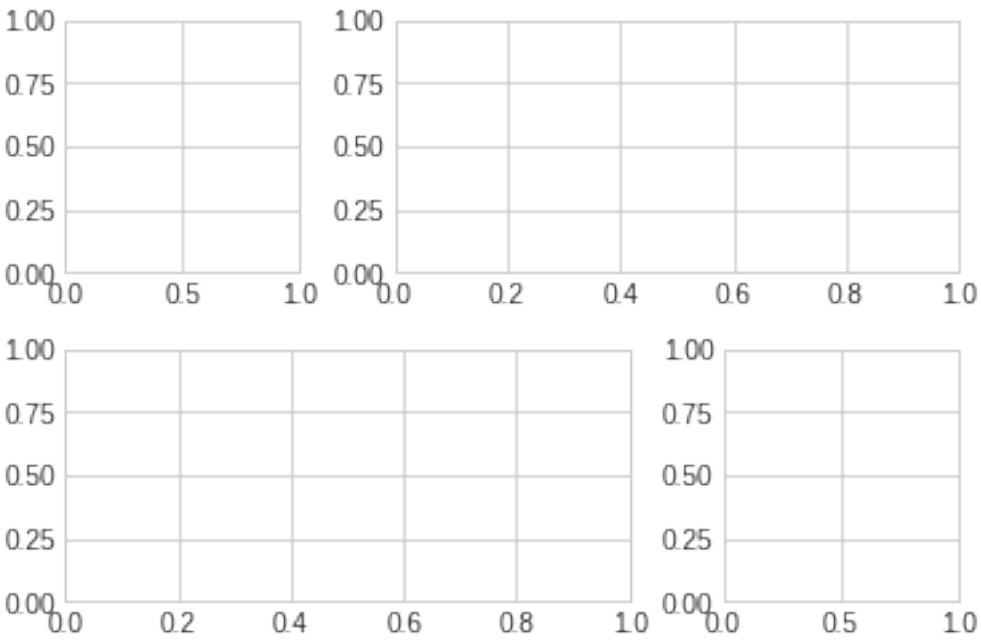
[32]: [<matplotlib.lines.Line2D at 0x7f59f39ae820>]



1.1.11 More complicated arrangements!! (similar to Bootstrap grid sys.)

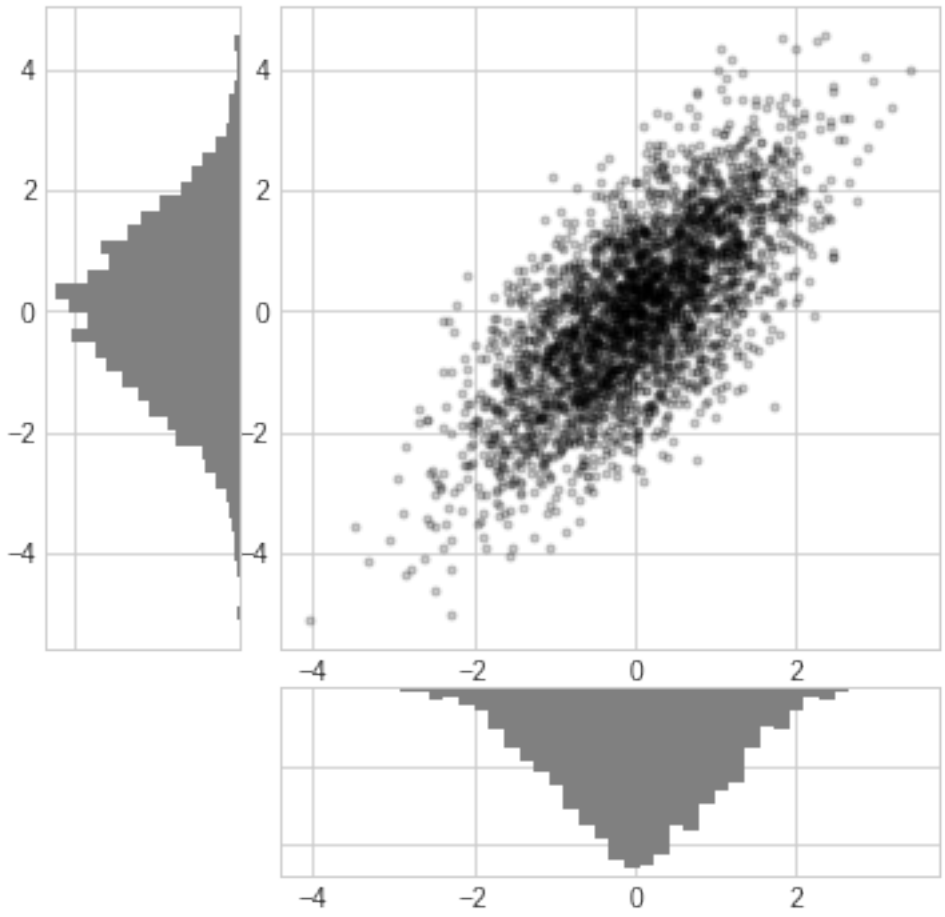
[33]:
```
grid = plt.GridSpec(2, 3, wspace=0.4, hspace=0.3)
plt.subplot(grid[0, 0])
plt.subplot(grid[0, 1:])
plt.subplot(grid[1, :2])
plt.subplot(grid[1, 2])
```

[33]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x7f59f389f130>
```



[34]:
```
##good example is to use when you want to show distribution
##along the x and y distribution

# Create some normally distributed data
mean = [0, 0]
cov = [[1, 1], [1, 2]]
x, y = np.random.multivariate_normal(mean, cov, 3000).T  #swap the axis, and
 ↪assign first sample to x, second sample to y

# Set up the axes with gridspec
fig = plt.figure(figsize=(6, 6))
grid = plt.GridSpec(4, 4, hspace=0.2, wspace=0.2)
main_ax = fig.add_subplot(grid[:-1, 1:])
y_hist = fig.add_subplot(grid[:-1, 0], xticklabels=[], sharey=main_ax)
x_hist = fig.add_subplot(grid[-1, 1:], yticklabels=[], sharex=main_ax)

# scatter points on the main axes
main_ax.plot(x, y, 'ok', markersize=3, alpha=0.2)

# histogram on the attached axes
x_hist.hist(x, bins = 40,
            orientation='vertical', color='gray')
x_hist.invert_yaxis()  #to flip the graph horizontally

y_hist.hist(y, bins = 40,
```

```
            orientation='horizontal', color='gray')
y_hist.invert_xaxis()  #to flip the graph vertically
```

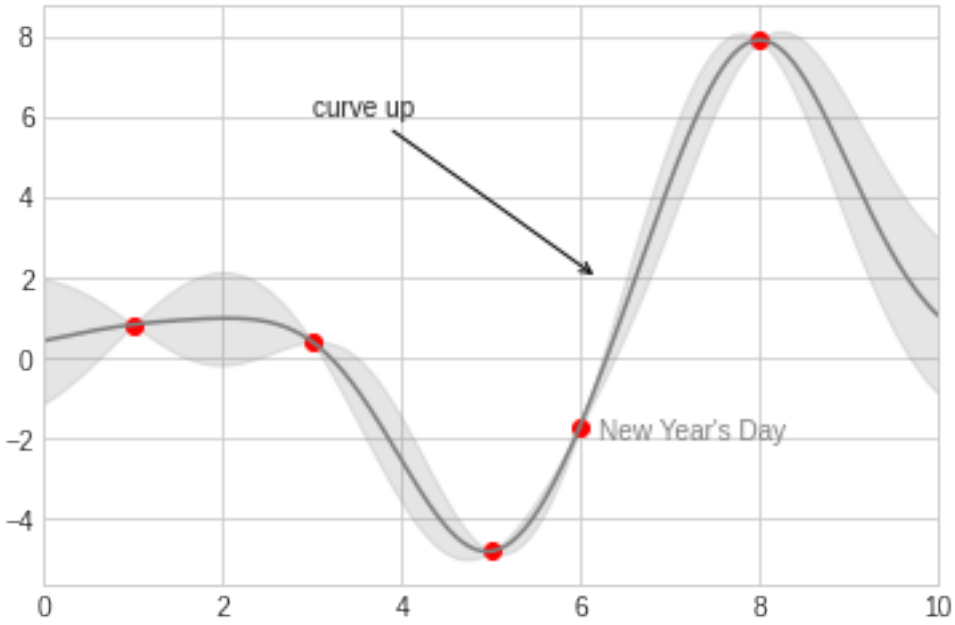


1.1.12 Annotation

[35]:
```
from sklearn.gaussian_process import GaussianProcessRegressor as GP

# define the model and draw some data
model = lambda x: x * np.sin(x)
xdata = np.array([1, 3, 5, 6, 8])
ydata = model(xdata)

# Compute the Gaussian process fit
gp = GP().fit(xdata[:, np.newaxis], ydata)

xfit = np.linspace(0, 10, 1000)
yfit, sigma = gp.predict(xfit[:, np.newaxis], return_std=True)
```

```
dyfit = 1.96 * sigma  # 95% confidence region

# Visualize the result
plt.plot(xdata, ydata, 'or')  #red circle
plt.plot(xfit, yfit, '-', color='gray')   #fitted by guassian

plt.fill_between(xfit, yfit - dyfit, yfit + dyfit,
                 color='gray', alpha=0.2)
plt.xlim(0, 10)

#add labels to the plot
style = dict(size = 10, color='gray')
plt.text(6.2, -2, "New Year's Day", **style)  #can simply put color
plt.annotate('curve up', xy=(6.2, 2), xytext=(3, 6),  #xytext ---> xy
             arrowprops=dict(arrowstyle="->"))

#there are many plot styles!
#publish-ready without any photoshop touch!  cool right?
#https://matplotlib.org/3.2.1/tutorials/text/annotations.html
```

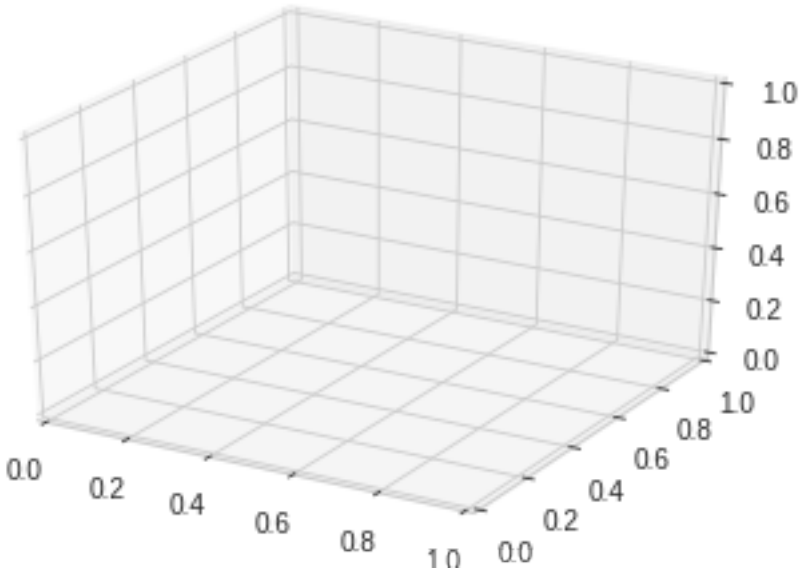
[35]: Text(3, 6, 'curve up')



1.1.13 3D Plotting

We can enable 3D plotting by importing the mplot3d library, which comes with your standard Matplotlib installation via pip. Just be sure that your Matplotlib version is over 1.0. Once this sub-module is imported, 3D plots can be created by passing the keyword projection="3d" to any of the regular axes creation functions in Matplotlib:

[36]:
```
from mpl_toolkits import mplot3d

import numpy as np
import matplotlib.pyplot as plt

fig = plt.figure()
ax = plt.axes(projection="3d")

plt.show()
```

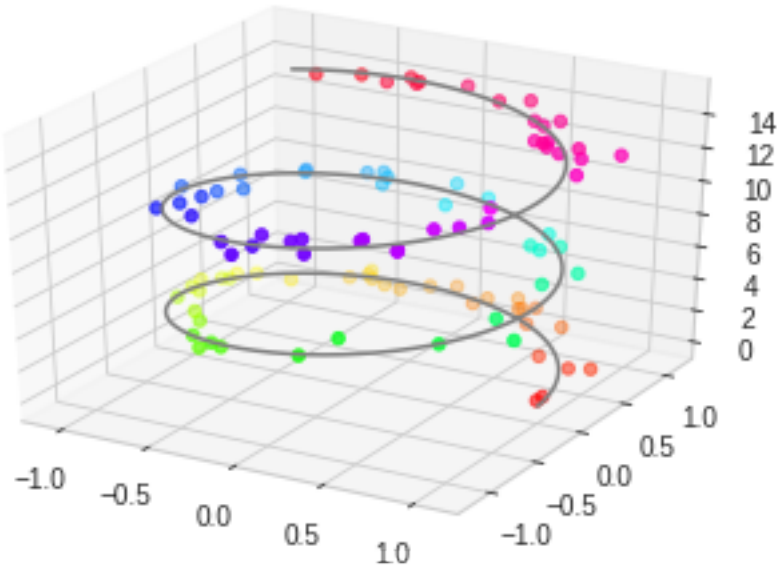


The 3D plotting functions are quite intuitive: instead of just scatter we call scatter3D , and instead of passing only x and y data, we pass over x, y, and z. All of the other function settings such as colour and line type remain the same as with the 2D plotting functions.

3D Scatter and Line Plots

[37]:
```
fig = plt.figure()
ax = plt.axes(projection="3d")

z_line = np.linspace(0, 15, 1000)
x_line = np.cos(z_line)
```

```
y_line = np.sin(z_line)
ax.plot3D(x_line, y_line, z_line, 'gray')

z_points = 15 * np.random.random(100)
x_points = np.cos(z_points) + 0.1 * np.random.randn(100)
y_points = np.sin(z_points) + 0.1 * np.random.randn(100)
ax.scatter3D(x_points, y_points, z_points, c=z_points, cmap='hsv');

plt.show()
```



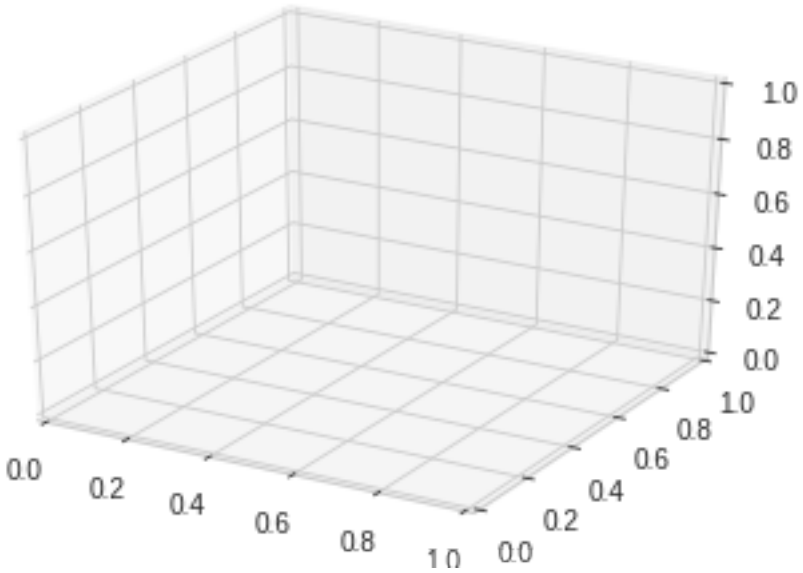
Surface Plots Surface plots can be great for visualising the relationships among 3 variables across the entire 3D landscape. They give a full structure and view as to how the value of each variable changes across the axes of the 2 others.

Constructing a surface plot in Matplotlib is a 3-step process.

1. First we need to generate the actual points that will make up the surface plot. Now, generating all the points of the 3D surface is impossible since there are an infinite number of them! So instead, we'll generate just enough to be able to estimate the surface and then extrapolate the rest of the points. We'll define the x and y points and then compute the z points using a function.

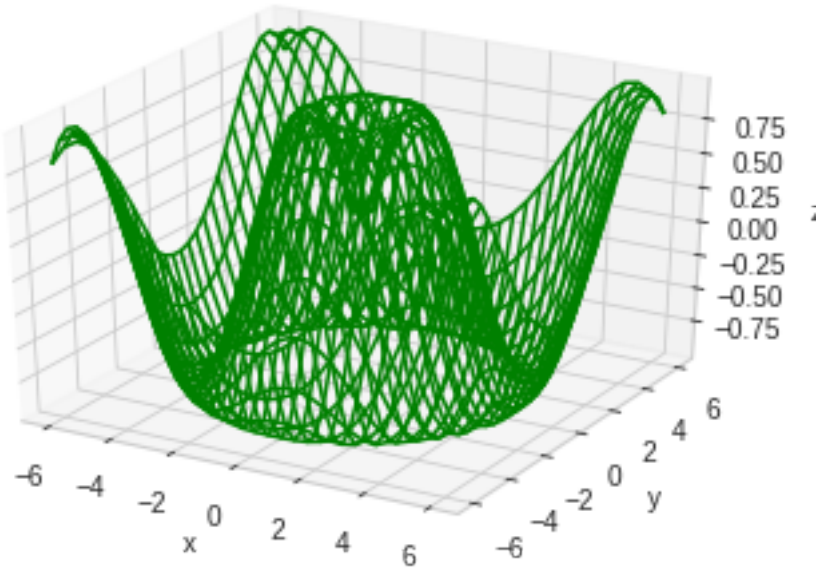
[38]:
```
fig = plt.figure()
ax = plt.axes(projection="3d")
def z_function(x, y):
    return np.sin(np.sqrt(x ** 2 + y ** 2))
```

```
x = np.linspace(-6, 6, 30)
y = np.linspace(-6, 6, 30)

X, Y = np.meshgrid(x, y)
Z = z_function(X, Y)
```



2. The second step is to plot a wire-frame — this is our estimate of the surface.

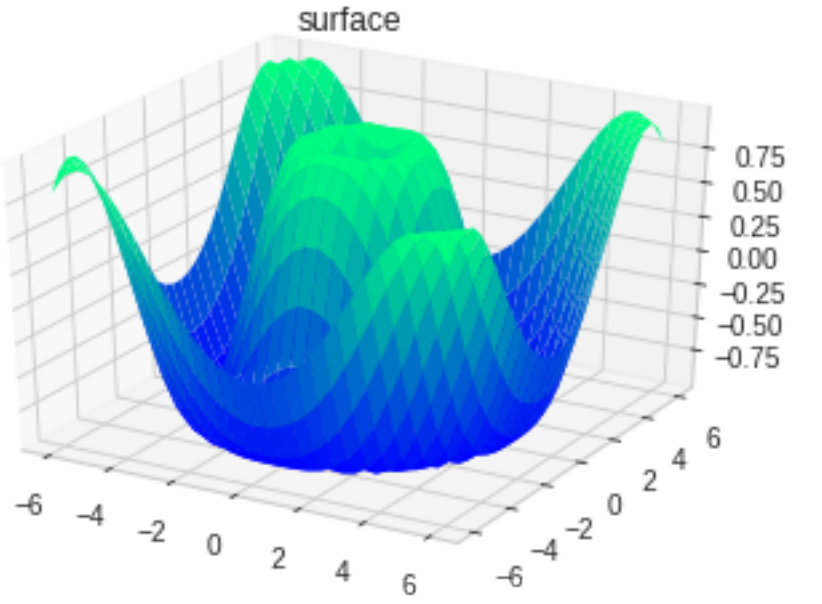
[39]:
```
fig = plt.figure()
ax = plt.axes(projection="3d")
ax.plot_wireframe(X, Y, Z, color='green')
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel('z')

plt.show()
```



3. Finally, we'll project our surface onto our wire-frame estimate and extrapolate all of the points.

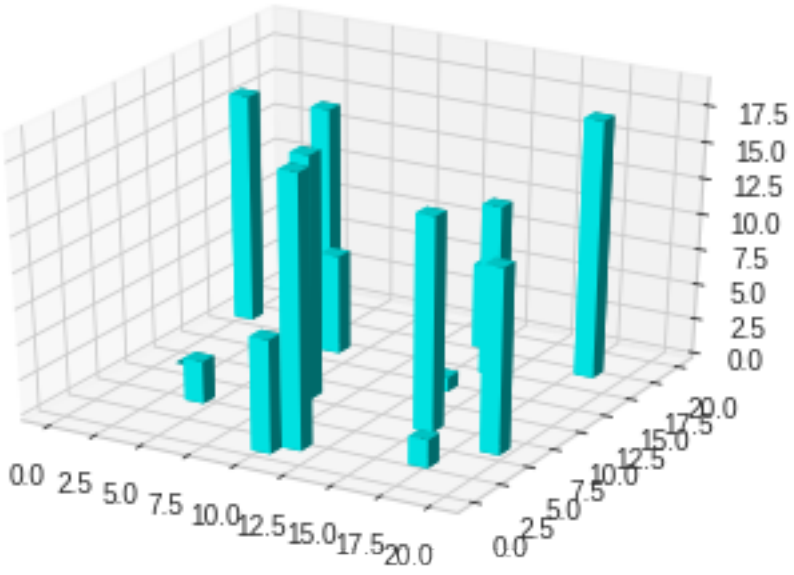
[40]:
```
ax = plt.axes(projection='3d')
ax.plot_surface(X, Y, Z, rstride=1, cstride=1,
                cmap='winter', edgecolor='none')
ax.set_title('surface')
```

[40]:
```
Text(0.5, 0.92, 'surface')
```



3D Bar Plots We'll select the z axis to encode the height of each bar; therefore, each bar will start at z = 0 and have a size that is proportional to the value we are trying to visualise. The x and y positions will represent the coordinates of the bar across the 2D plane of z = 0. We'll set the x and y size of each bar to a value of 1 so that all the bars have the same shape.

[41]:
```
import random

fig = plt.figure()
ax = plt.axes(projection="3d")

num_bars = 15
x_pos = random.sample(list(range(20)), num_bars)
y_pos = random.sample(list(range(20)), num_bars)
z_pos = [0] * num_bars
x_size = np.ones(num_bars)
y_size = np.ones(num_bars)
z_size = random.sample(list(range(20)), num_bars)

ax.bar3d(x_pos, y_pos, z_pos, x_size, y_size, z_size, color='aqua')
plt.show()
```



1.1.14 Date Tick Labels

You can use .set_major_locator, .set_major_formatter, .set_minor_locator to format the ticks

[42]:
```
import os
os.environ['http_proxy'] = 'http://192.41.170.23:3128'
os.environ['https_proxy'] = 'http://192.41.170.23:3128'

import numpy as np
import matplotlib.pyplot as plt
import matplotlib.dates as mdates

plt.rcParams.update({'xtick.labelsize': 10, 'xtick.major.size': 10,
                     'xtick.minor.size': 5})

#to get all the avail keys:
# plt.rcParams.keys

years = mdates.YearLocator()   # every year
months = mdates.MonthLocator()  # every month
years_fmt = mdates.DateFormatter('%Y')

# Load a numpy structured array from yahoo csv data with fields date, open,
# close, volume, adj_close from the mpl-data/example directory.  This array
# stores the date as an np.datetime64 with a day unit ('D') in the 'date'
# column.
from pandas_datareader import data

goog = data.DataReader('GOOG', start='2004', end='2016',
                       data_source='yahoo')

goog = goog['Adj Close']

fig, ax = plt.subplots(figsize=(10, 10))
ax.plot(goog)

# format the ticks
ax.xaxis.set_major_locator(years)
ax.xaxis.set_major_formatter(years_fmt)
ax.xaxis.set_minor_locator(months)

# format the coords message box
ax.format_xdata = mdates.DateFormatter('%Y-%m-%d')
ax.format_ydata = lambda x: '$%1.2f' % x  # format the price.
ax.grid(True)

# rotates and right aligns the x labels, and moves the bottom of the
```

```
# axes up to make room for them
fig.autofmt_xdate()

plt.show()
```

1.1.15 Tables

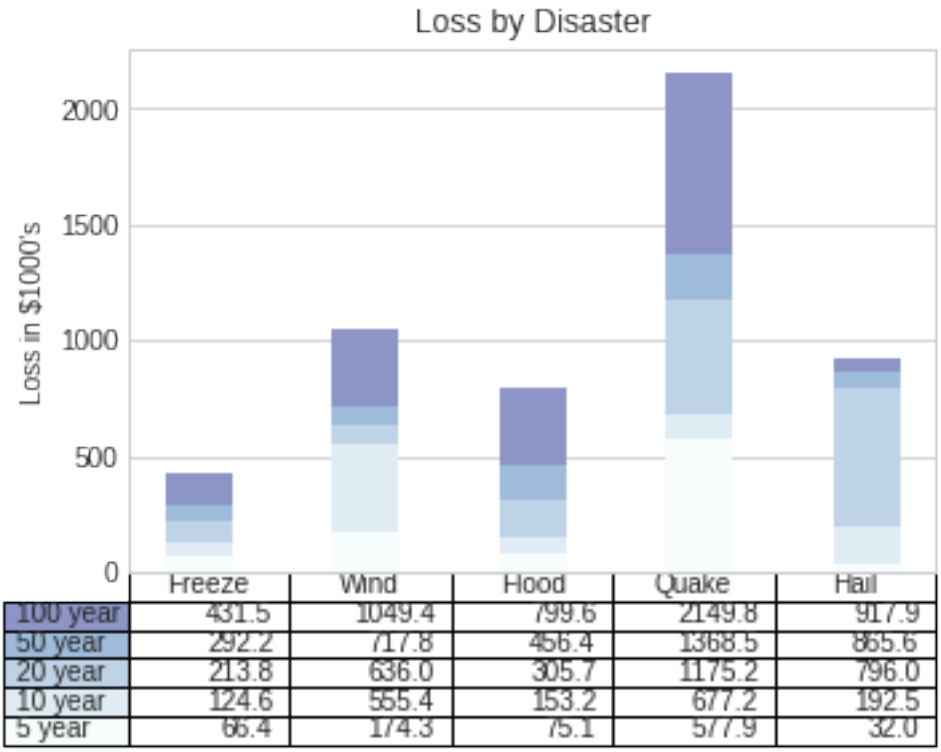
We can use .table to embed some table information into the graph

[43]:
```
import numpy as np
import matplotlib.pyplot as plt


data = [[ 66386, 174296,  75131, 577908,  32015],
        [ 58230, 381139,  78045,  99308, 160454],
```

```
        [ 89135,  80552, 152558, 497981, 603535],
        [ 78415,  81858, 150656, 193263,  69638],
        [139361, 331509, 343164, 781380,  52269]]

columns = ('Freeze', 'Wind', 'Flood', 'Quake', 'Hail')
rows = ['%d year' % x for x in (100, 50, 20, 10, 5)]

values = np.arange(0, 2500, 500)
value_increment = 1000

# Get some pastel shades for the colors
colors = plt.cm.BuPu(np.linspace(0, 0.5, len(rows)))
n_rows = len(data)

#position of the bars
index = np.arange(len(columns)) + 0.3

bar_width = 0.4

# Initialize the vertical-offset for the stacked bar chart.
y_offset = np.zeros(len(columns))

print(y_offset)

# Plot bars and create text labels for the table
cell_text = []
for row in range(n_rows):
    plt.bar(index, data[row], bar_width, bottom=y_offset, color=colors[row])
    y_offset = y_offset + data[row]
    print(['%1.1f' % (x / 1000.0) for x in y_offset])
    cell_text.append(['%1.1f' % (x / 1000.0) for x in y_offset])
# Reverse colors and text labels to display the last value at the top.
colors = colors[::-1]
cell_text.reverse()

# Add a table at the bottom of the axes
the_table = plt.table(cellText=cell_text,
                      rowLabels=rows,
                      rowColours=colors,
                      colLabels=columns,
                      loc='bottom')

# Adjust layout to make room for the table:
plt.subplots_adjust(left=0.2, bottom=0.2)

plt.ylabel("Loss in ${0}'s".format(value_increment))
plt.yticks(values * value_increment, ['%d' % val for val in values])
```

```
plt.xticks([])
plt.title('Loss by Disaster')

plt.show()
```

```
[0. 0. 0. 0. 0.]
['66.4', '174.3', '75.1', '577.9', '32.0']
['124.6', '555.4', '153.2', '677.2', '192.5']
['213.8', '636.0', '305.7', '1175.2', '796.0']
['292.2', '717.8', '456.4', '1368.5', '865.6']
['431.5', '1049.4', '799.6', '2149.8', '917.9']
```



	Freeze	Wind	Hood	Quake	Hail
100 year	431.5	1049.4	799.6	2149.8	917.9
50 year	292.2	717.8	456.4	1368.5	865.6
20 year	213.8	636.0	305.7	1175.2	796.0
10 year	124.6	555.4	153.2	677.2	192.5
5 year	66.4	174.3	75.1	577.9	32.0

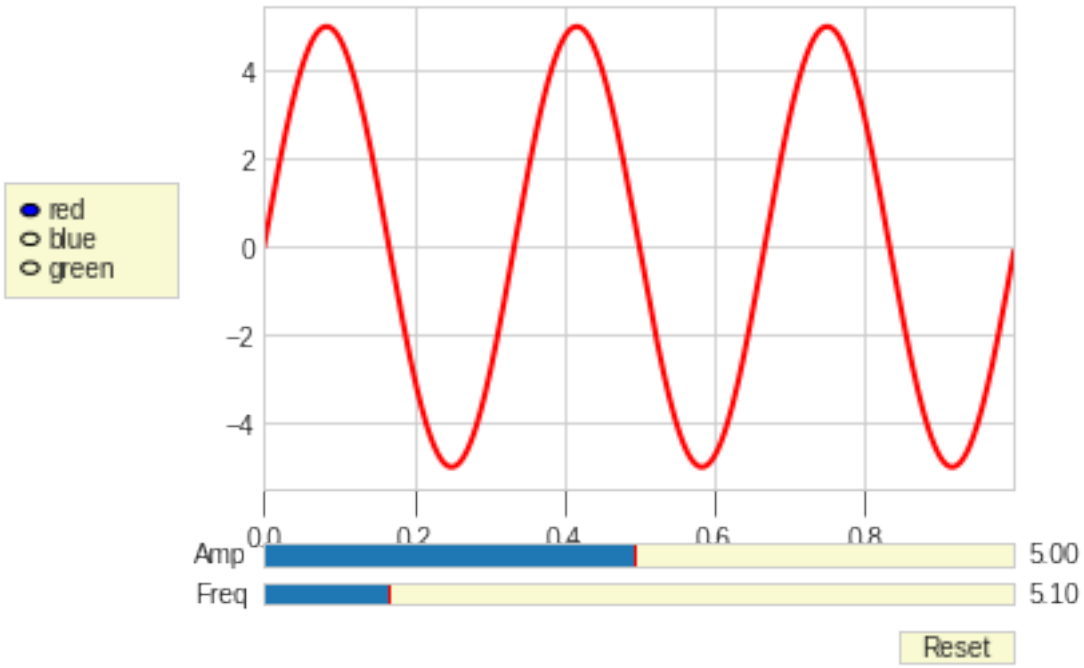
1.1.16 GUI

Using event-based triggers such as on_changed | on_clicked from the matplotlib.widgets, we can interact with visualizations.

[44]:
```
# %matplotlib notebook  --> put this if you already install tk

import numpy as np
import matplotlib.pyplot as plt
from matplotlib.widgets import Slider, Button, RadioButtons
```

```
fig, ax = plt.subplots()
plt.subplots_adjust(left=0.25, bottom=0.25)
t = np.arange(0.0, 1.0, 0.001)
a0 = 5
f0 = 3
delta_f = 5.0
s = a0 * np.sin(2 * np.pi * f0 * t)
l, = plt.plot(t, s, lw=2)
ax.margins(x=0)

axcolor = 'lightgoldenrodyellow'
axfreq = plt.axes([0.25, 0.1, 0.65, 0.03], facecolor=axcolor)
axamp = plt.axes([0.25, 0.15, 0.65, 0.03], facecolor=axcolor)

sfreq = Slider(axfreq, 'Freq', 0.1, 30.0, valinit=f0, valstep=delta_f)
samp = Slider(axamp, 'Amp', 0.1, 10.0, valinit=a0)


def update(val):
    amp = samp.val
    freq = sfreq.val
    l.set_ydata(amp*np.sin(2*np.pi*freq*t))
    fig.canvas.draw_idle()


sfreq.on_changed(update)
samp.on_changed(update)

resetax = plt.axes([0.8, 0.025, 0.1, 0.04])
button = Button(resetax, 'Reset', color=axcolor, hovercolor='0.975')

def reset(event):
    sfreq.reset()
    samp.reset()
button.on_clicked(reset)

rax = plt.axes([0.025, 0.5, 0.15, 0.15], facecolor=axcolor)
radio = RadioButtons(rax, ('red', 'blue', 'green'), active=0)

def colorfunc(label):
    l.set_color(label)
    fig.canvas.draw_idle()
radio.on_clicked(colorfunc)

# Initialize plot with correct initial active value
colorfunc(radio.value_selected)
```

```
plt.show()
```



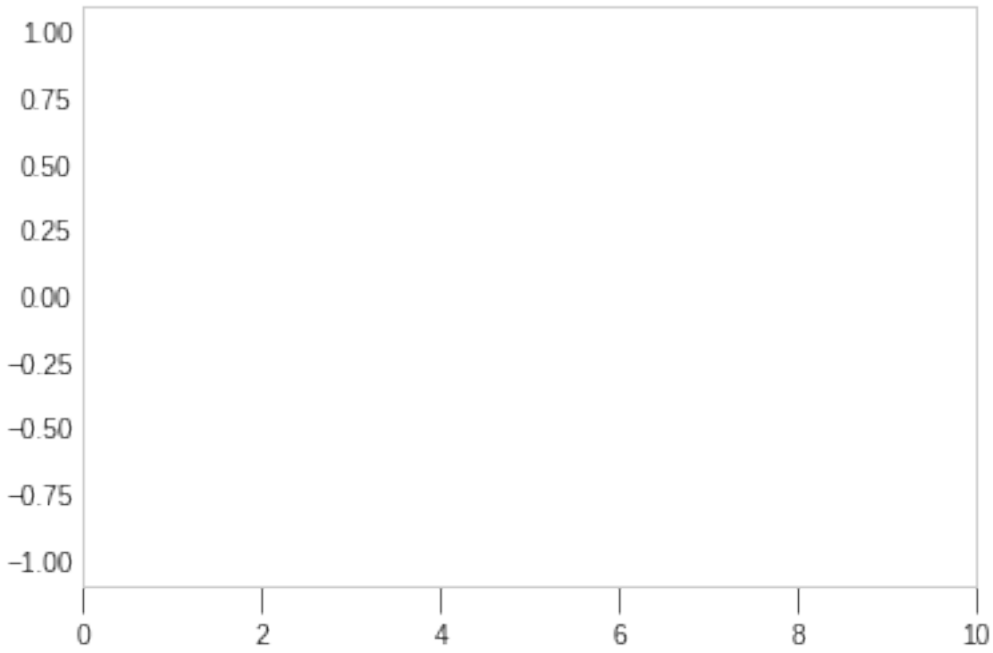
1.1.17 Animation

We can use matplotlib.animation to create some nice animation

[49]:
```
import matplotlib.pyplot as plt
import matplotlib.animation as animation
import itertools

def data_gen():
    for cnt in itertools.count():
        t = cnt / 10
        yield t, np.sin(2*np.pi*t) * np.exp(-t/10.)


def init():
    ax.set_ylim(-1.1, 1.1)
    ax.set_xlim(0, 10)
    del xdata[:]
    del ydata[:]
    line.set_data(xdata, ydata)
```

```
    return line,

fig, ax = plt.subplots()
line, = ax.plot([], [], lw=2)
ax.grid()
xdata, ydata = [], []


def run(data):
    # update the data
    t, y = data
    xdata.append(t)
    ydata.append(y)
    xmin, xmax = ax.get_xlim()

    if t >= xmax:
        ax.set_xlim(xmin, 2*xmax)
        ax.figure.canvas.draw()
    line.set_data(xdata, ydata)

    return line,

ani = animation.FuncAnimation(fig, run, data_gen, interval=10, init_func=init)

from IPython.display import HTML
HTML(ani.to_html5_video())
```

[49]: <IPython.core.display.HTML object>


1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
0
2
4
6
8
10

[ ]: